User's Guide

DCAM Video Processing Development Kit

(using micro-line® C6412Compact)

Contents

1 PREFACE 6

1.1 Document Organization 6

1.2 Documentation Overview 6

1.3 Notational conventions 6

1.4 Trademarks 7

1.5 History 8

2 KIT OVERVIEW 9

2.1 Hardware 10
2.1.1 C6412Compact 10
2.1.2 Power Supply Board 10

2.2 Software 11
2.2.1 Application Examples 11
2.2.2 DCAM Frame Capture API 11
2.2.3 DCAM Camera API 11
2.2.4 Software Streaming Libraries 12
2.2.5 Board Library 12
2.2.6 Flash File System 12
2.2.7 FPGA Configuration Files 12

3 GETTING STARTED 13

4 SOFTWARE DEVELOPMENT 16

4.1 Installation 16

4.2 Required Tools 16
4.2.1 Code Composer Studio Setup 16

4.3 Development Flow 17

4.4 Required DSP/BIOS Settings and Objects 18
4.4.1 Memory Layout 18
4.4.2 Clock Rate 18
4.4.3 Cache Setup 19
4.4.4 Heap Setup 19
4.4.5 Section location 19
4.4.6 Interrupts 20
4.4.7 Tasks 20
4.4.8 Semaphores 20
4.4.9 Resource Locks 21
4.4.10 Frame Buffers 21
4.4.11 Idle Tasks 21

4.5 Loading Software Using a JTAG Emulator 21

4.6 Debugging Software Using a JTAG Emulator 22

4.7 Storing Application Software in Flash Memory 22

4.8 Storing FPGA Code in Flash Memory 23

5 APPLICATION EXAMPLES 25

5.1 VP_example 25
5.1.1 Application structure 25
5.1.2 System initialization 26
5.1.3 Main Control Task 28
5.1.4 Frame Processing Task 32
5.1.5 Image Processing 34
5.1.6 Output-side Setup 35
5.1.7 Output-side Callbacks 36
5.1.8 Resource Usage 36
5.1.9 Performance 36
5.1.10 Modifying for In-place Processing 37
5.1.11 Control and Data Flow 37

6 SYSTEM-SPECIFIC DATA TYPES 39

6.1 Incoming Streaming Statistics 39

6.2 Outgoing Streaming Statistics 39

7 SYSTEM-SPECIFIC FUNCTIONS 40

7.1 API Callback Queue Functions 40
7.1.1 InitCallQ 40
7.1.2 YieldToCallQ 40

7.2 Incoming Isochronous Streaming Functions 41
7.2.1 Incoming Isochronous Streaming Data Types 41
7.2.2 dcaminIsoInit 42
7.2.3 dcaminIsoAllocateResources 42
7.2.4 dcaminIsoListen 43
7.2.5 dcaminIsoGetBufferState 45
7.2.6 dcaminIsoReleaseBuffer 45
7.2.7 dcaminIsoStop 46
7.2.8 dcaminIsoDeallocateResources 46
7.2.9 dcaminIsoGetStatistics 47
7.2.10 dcaminIsoGetLastError 47

7.3 Outgoing Isochronous Streaming Functions 47
7.3.1 Outgoing Isochronous Streaming Data Types 48
7.3.2 dcamoutIsoInit 49
7.3.3 dcamoutIsoTalk 49
7.3.4 dcamoutIsoGetBufferState 50
7.3.5 dcamoutIsoLockBuffer 51

7.3.6 dcamoutIsoSubmitBuffer 51
7.3.7 dcamoutIsoStop 52
7.3.8 dcamoutIsoGetStatistics 52
7.3.9 dcamoutIsoGetLastError 52

7.4 Status Codes 53
7.4.1 DCAMIN_ERROR_INVALID_PARAMETER 53
7.4.2 DCAMIN_ERROR_ISO_RESOURCE 53
7.4.3 DCAMIN_ERROR_ISO_START 54
7.4.4 DCAMIN_ERROR_ISO_STOP 54
7.4.5 DCAMIN_ERROR_ISO_BUFSTAT 54
7.4.6 DCAMIN_ERROR_ISO_BUFFER 54
7.4.7 DCAMIN_ERROR_ISO_STATISTICS 54
7.4.8 DCAMOUT_ERROR_ISO_INIT 54
7.4.9 DCAMOUT_ERROR_ISO_START 54
7.4.10 DCAMOUT_ERROR_ISO_STOP 54
7.4.11 DCAMOUT_ERROR_ISO_BUFSTAT 54
7.4.12 DCAMOUT_ERROR_ISO_BUFFER 55
7.4.13 DCAMOUT_ERROR_ISO_STATISTICS 55
7.4.14 DCAMOUT_ERROR_UNSUPPORTED_ISO_CHANNEL 55
7.4.15 DCAMOUT_ERROR_INVALID_ISO_PACKET_SIZE 55
7.4.16 DCAMOUT_ERROR_UNSUPPORTED_ISO_SPEED 55

8 FPGA DEVELOPMENT SUPPORT 56

9 LIST OF ABBREVIATIONS USED IN THIS DOCUMENT 57

10 LITERATURE REFERENCES 58

List of Tables

Table 1: Function callability table for incoming isochronous streaming 41
Table 2: Function callability table for outgoing isochronous streaming 48
Table 3: Status codes summary 53

List of Figures

DCAM Video Processsing Development Kit block diagram (without camera) 9
DCAM Video Processing Kit connectors and control elements 10
Session example: storing application code to flash memory 23
Session example: storing FPGA code to flash memory 24
VP_example output (still image vs. moving) 25

1 Preface

This document describes the DCAM Video Processing Development Kit for the C6412Compact. The DCAM Video Processing Development Kit is a complete platform for embedded image processing with industrial IEEE1394 cameras.

1.1 Document Organization

This document is organized as follows:

- Chapter 2 gives an overview of the development kit.
- Chapter 3 shows how to do the first steps with the kit.
- Chapter 4 gives an introduction to software development on the C6412Compact.
- Chapter 5 describes the application examples.
- Chapter 6 documents data types that are not covered by [3] and [4].
- Chapter 7 documents functions that are not covered by [3] and [4].
- Chapter 8 introduces the FPGA development option.
- Chapter 9 explains the abbreviations that are used throughout this document.
- Chapter 10 lists documents that contain further information.

1.2 Documentation Overview

This chapter lists the documentation from Orsys that is shipped together with the DCAM Video Processing Development Kit. Further documents from other vendors are listed in chapter 10 and are referenced throughout the document in square brackets.

micro-line Power Supply Kit (`Power_Supply.pdf`):
Describes the power supply board.

C6412Compact Hardware Reference Guide [8] (`C6412CPT_hrg.pdf`):
Describes the hardware and the basic features of the C6412Compact. Use this manual to get an overview of the board hardware and as a reference for technical details of the board.

C6412Compact micro-line® busmaster BSP User's Guide [6] (`C6412CPT_ml_bm_ug.pdf`):
Describes the micro-line® busmaster Board Support Package (BSP), which is used for operation of the IEEE1394 interfaces. Use this manual to check for additional functions and signals.

DCAM Frame Capture API User's Guide [1] `(DCAM_FC_API_ug.pdf):`
Describes the DCAM Frame Capture API, which is used on the input side for controlling connected cameras and getting image data.

DCAM Camera API User's Guide [4] `(DCAM_Camera_API_ug.pdf):`
Describes the DCAM Camera API, which is used on the output side to handle incoming control requests and to transmit processed image data.

1.3 Notational conventions

Names of registers, bit fields and single bits are written in capital letters.
Example: HWCFG

Names of signals are also given in capital letters; active low signals are marked with a '/' at the beginning of the name.
Example: /RESETIN

Configuration parameters, function names, path names and file names are written in *italic* typeface.
Example: *dev_id*

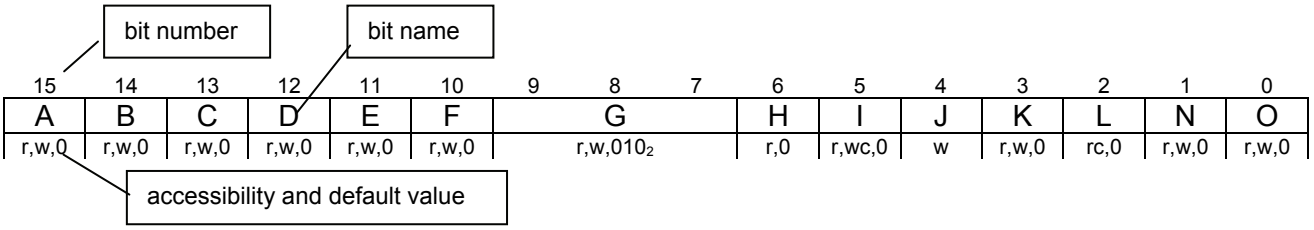
Source code examples are given in a small, fixed-width typeface.
Example: `int a = 10;`

Menus and commands from menus and submenus are enclosed in double-quotes. Example:
Create a new project using the "Create Project..." command from the "File" menu.

The members of a bit field or a group of signals are numbered starting at zero, which is the least significant bit.
Example: CFG[4:0] identifies a group of five signals, where CFG0 is the least significant bit and CFG4 is the most significant bit.

If necessary, numbers are represented with a suffix that specifies their base.
Example: $12AB_{16}$ is a hexadecimal number (base 16 = hexadecimal) and is equal to 4779_{10}.

The bit fields of a register are displayed with the most significant bit to the left. Below each bit field is a description of its read / write accessibility and its default value:


bit number
bit name
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
A B C D E F G H I J K L N O
r,w,0 r,w,0 r,w,0 r,w,0 r,w,0 r,w,0 r,w,010₂ r,0 r,wc,0 w r,w,0 rc,0 r,w,0 r,w,0
accessibility and default value

legend:

r	bit is readable
rc	this bit is cleared after a read
r,w	bit is readable and writeable, reading yields the previously written value unless otherwise specified.
w	bit is writeable, read value is undefined
wc	writing a 1 to this bit clears it
w,0	bit is write-only, reading always yields 0.
0	default value

1.4 Trademarks

TI, Code Composer, DSP/BIOS and TMS320C6000 are registered trademarks of Texas Instruments.

Microsoft® and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Hyperterminal is a trademark of Hilgraeve Inc.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

1.5 History

Revision	Changes
1.0	First release
1.1	Added functions *dcaminGetLastError* and *dcamoutGetLastError*.

2 Kit Overview

The DCAM Video Processing Kit consists of the following components:

- a micro-line C6412Compact DSP/FPGA board with two IEEE1394 interfaces
- a micro-line power supply carrier board for convenient operation on the desk during development
- a Unibrain Fire-i camera
- IEEE1394 and RS-232 cables
- an AC adapter
- a distribution CD with
  - C6412Compact board library
  - C6412Compact micro-line busmaster BSP
  - Software streaming libraries
  - DCAM frame capture API (input side of video processing)
  - DCAM camera API (output side of video processing)
  - an application example
  - documentation of the above listed components

IEEE1394 interface 1 of the C6412Compact is used as an image input and is connected to an IIDC camera. User-defined image processing takes the incoming image data, processes it and sends it to the second IEEE1394 interface, which is used as image output. A camera controller (typically a PC) connected to the second IEEE1394 interface recognizes the C6412Compact as an IIDC compliant camera and receives the processed images.
Both, input and output are compliant to the IIDC 1394-based Digital Camera Specification V1.30.
For the input side, multiple cameras are supported, whereas the output side can only implement a single image data stream.

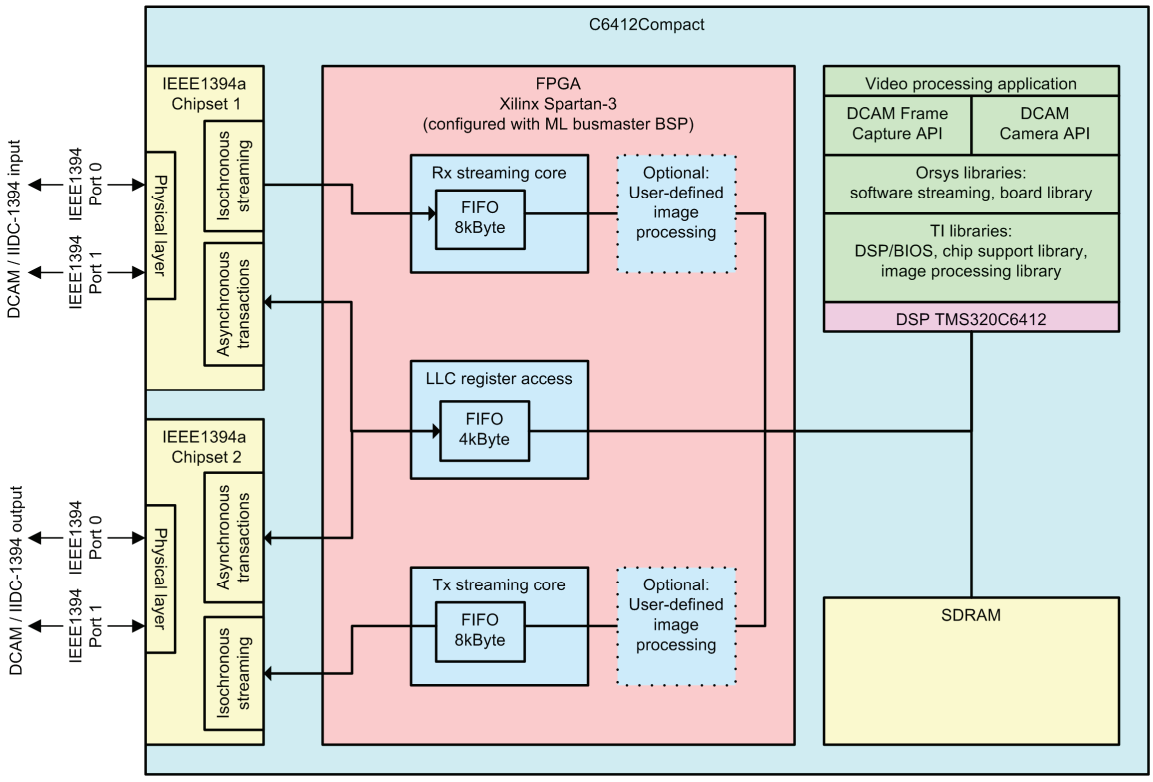


Figure 1: DCAM Video Processsing Development Kit block diagram (without camera)

2.1 Hardware

The hardware of the DCAM Video Processing Development Kit comprises a C6412Compact board mounted on a power supply board and an IIDC 1394 compliant digital camera.

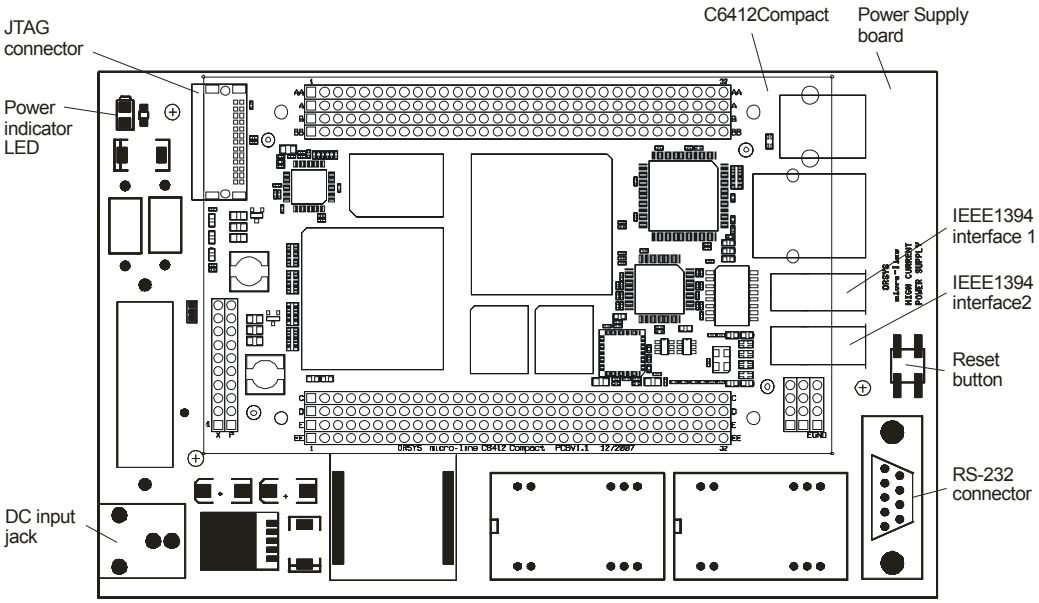


Figure 2: DCAM Video Processing Kit connectors and control elements

2.1.1 C6412Compact

The C6412Compact is a high performance DSP / FPGA board suitable for high speed data processing. Key features for video processing are:

- 720 MHz CPU clock (600 MHz available for low-cost versions)
- 64 MB or 128 MB SDRAM allow implementation of large image buffers
- Two independent IEEE1394 interfaces allow operation at the full isochronous data rate of 32,768,000 bytes/s for a single camera[1].
- FPGA-based, FIFO-buffered isochronous streaming minimizes CPU overhead for data transfers
- A 120 MHz, 64-bit wide bus interface between DSP, FPGA and SDRAM provides high-bandwidth data transfers with minimum overhead.
- The FPGA development option provides the possibility of FPGA-based image processing
- Direct access to the DSP over the JTAG connector for code upload, debugging and real-time data exchange using standard development tools.
- On-board Flash File System for non-volatile storage of application code and FPGA code.

For use with the DCAM Video Processing Kit, the C6412Compact is configured for cable power supply output on interface 1. This allows to supply a camera (connected to IEEE1394 interface 1) over the IEEE1394 cable.
The hardware of the C6412Compact and its interfaces are described in the C6412Compact Hardware Reference Guide [8].

2.1.2 Power Supply Board

The power supply board acts as a mechanical carrier for the C6412Compact to be operated on the desk during development. It provides all necessary power supply and connectors. The power

[1] For multiple cameras the maximum data rate is determined by available bus bandwidth, which is up to 20% higher, but depends on system topology.

supply board provides a main power supply of +3.3 V for the C6412Compact as well as (optionally) two galvanically isolated voltages to be used by peripheral boards (e.g. A/D converter boards). The power supply board is described in [1]. For use with the DCAM Video Processing Kit, the power supply board is configured provide IEEE1394 cable power to the C6412Compact. This allows connected cameras to be supplied from the kit.

2.2 Software

The software of the DCAM Video Processing Development Kit consists of

- application examples
- the DCAM Frame Capture API
- the DCAM Camera API
- Software streaming libraries
- the board library
- Flash File System utilities
- FPGA configuration files

Usage of the DCAM libraries requires DSP/BIOS (see [11]), version 5.31.02 or higher for system timing and interprocess synchronization. If a non-BIOS version is required, please contact Orsys. Accesses to DSP on-chip hardware are done using TI's chip support library (see [13]).

2.2.1 Application Examples

The DCAM Video Processing Development Kit contains one application example of a video processing application and some examples that illustrate the programming of the C6412Compact using the board library. These application examples can be used as starting points for own applications. The video processing example is described in chapter 5, whereas the board library examples are described in [8].

2.2.2 DCAM Frame Capture API

The DCAM Frame Capture API supports digital cameras that are compliant to the 1394-based Digital Camera Specification Version 1.30. It provides the required functionality to control the digital cameras and capture isochronous image data transmitted by them. The captured image data can then be processed by application software.
The API locates digital cameras connected to the IEEE1394 bus. A handle to the camera is assigned when opening a specific camera. Then, the camera is accessed using this handle. Typically, video modes and features of the camera are queried and modified and then the camera is started. The DCAM frame capture API is described in [3].

2.2.3 DCAM Camera API

The DCAM Camera API adds DCAM / IIDC 1394 digital camera functionality to the C6412Compact so that the C6412Compact is recognized as a digital camera by a camera controller, such as a PC running Windows XP. The DCAM Camera API handles all IEEE1394 communication and IIDC 1394 protocol handling.
The application sets up the digital camera according to the video modes and features that are supported. The video modes and features can then be queried and controlled over the IEEE1394 bus. IEEE1394 accesses to the mapped register sets are transformed into user callback functions. With help of these callback functions the specific behavior of the system is implemented by an application.
Normally frame capture software running on a PC (such as "Generic 1394 Desktop Camera") starts the camera. The application is informed by a user callback. The application then starts the connected digital camera(s) with the Frame Capture API. The received image data is processed by the DSP and the resulting images are sent with the Camera API to the PC. The DCAM camera API is described in [4].

2.2.4 Software Streaming Libraries

Both, the DCAM Frame Capture API and the DCAM Camera API use the software streaming libraries for image data transfer. This software package implements EDMA-based transfers between the frame buffers and the streaming core FIFOs in the FPGA. Events are generated whenever a buffer has been received (DCAM Frame Capture API) or has been transmitted (DCAM Camera API). Direct access to the software streaming libraries is usually not necessary, but for special cases, the isochronous transfer functions of the DCAM APIs can be bypassed by directly accessing the software streaming libraries. The software streaming package is described in [5]

2.2.5 Board Library

The board library is a collection of low level drivers that allow to access hardware components of the C6412Compact, such as Flash File System, RS-232 interface, temperature sensor, etc. This facilitates programming of the C6412Compact. The board library is described in [8].

2.2.6 Flash File System

The Flash File System (FFS) is integral part of the C6412Compact. It is already pre-installed when the C6412Compact is shipped. Therefore, after power on, the C6412Compact boots the FFS. See also [8] for a description of the boot process. At system startup, the FFS manages loading of

- an auto-start application (DSP software)
- the on-board FPGA
- external FPGAs.

Application and FPGA code can be stored in flash memory by using the PC Flash File System utilities. The Flash File System is documented in a separate user's guide, see [2].
For the DCAM Video Processing Development Kit, the FFS contains 3 files

- a default application, *toggle_led*, which is included in each C6412Compact that is shipped
- application example *VP_example*, marked as auto-start application
- FPGA code of the micro-line® busmaster BSP (*ml_bm*). This file is not marked for auto-load to prevent potential problems with customized hardware. However, application software can still configure the FPGA from flash memory using board library functions.

2.2.7 FPGA Configuration Files

The DCAM Video Processing Development Kit requires the FPGA to be configured with the micro-line© Busmaster Board Support Package (BSP). This BSP allows access to both IEEE1394 chipsets and. The micro-line© busmaster BSP is described in [6].

3 Getting Started

The C6412Compact DCAM Video Processing Development Kit is shipped with the application example *VP_example* already stored in flash memory. This allows doing the very first steps with the kit without any development tools required. The example looks for a suitable camera on IEEE1394 interface 1 and configures it to Format 0, Mode 1, Rate 4, which is 320x240 YUV(4:2:2) 30fps. This setting allows to use the example with the built-in features of Windows XP. If a suitable camera is present, the image processing task compares each picture with the preceding picture and highlights differences by dimming unmodified pixels. The processed image is available on IEEE1394 interface 2. Debug messages are output on the RS-232 interface. During image processing, the red LED is turned on as an optical activity indicator.

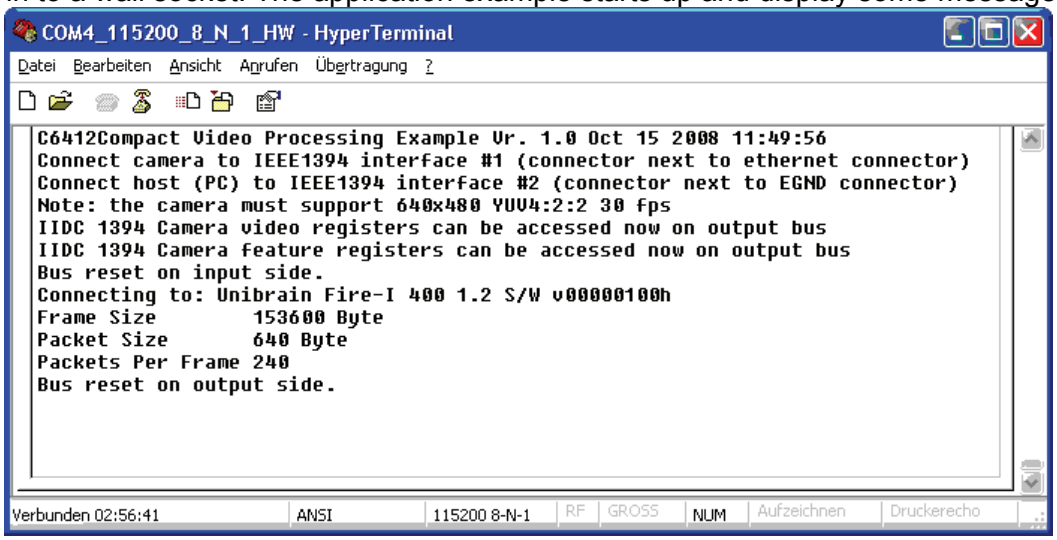
Below is a procedure for doing the first steps with the DCAM Video Processing Development Kit.

Required items:

- the C6412Compact DCAM Video Processing Development Kit
- a PC with
  - Windows XP
  - an IEEE1394 interface
  - a terminal program, such as HyperTerminal[2]

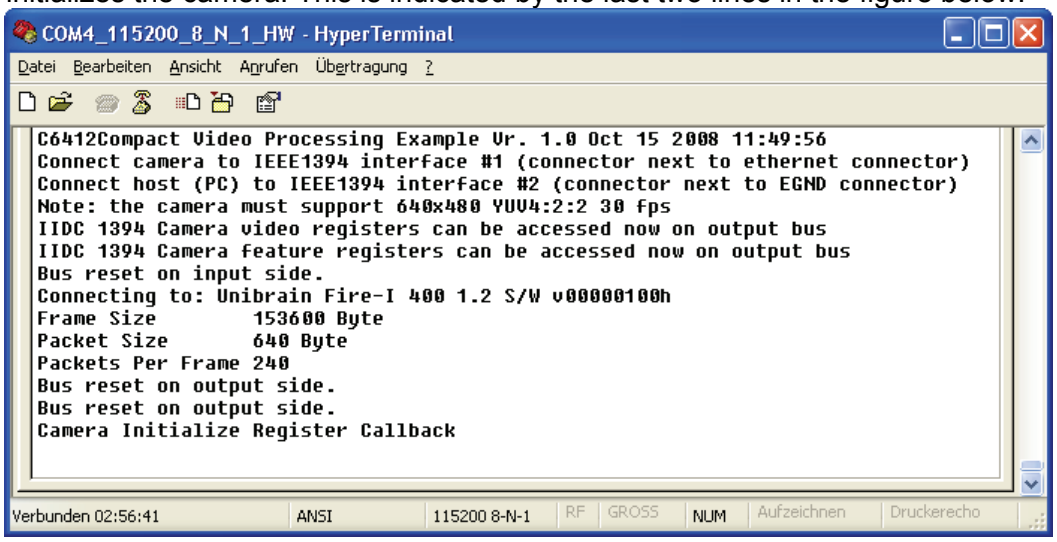
Steps to be performed:

- For viewing the debug messages, start a terminal program, such as "HyperTerminal" on the PC and set up the terminal program to 115200 bits per second, 8 data bits, no parity, 1 stop bit and hardware flow control.
- Connect the kit to the PC using the RS-232 cable. Please note that RS-232 output must be enabled to run this example[2]. In case of problems, please refer to the RS-232 wiring in [9]
- Connect the digital camera to the C6412Compact, IEEE1394 interface 1 using an IEEE1394 cable. Connector location is shown in Figure 2.
- Connect the C6412Compact, IEEE1394 interface 2 to the PC using an IEEE1394 cable
- Supply the kit with power, i.e. connect the AC adapter to the power supply board and plug it in to a wall socket. The application example starts up and display some messages:

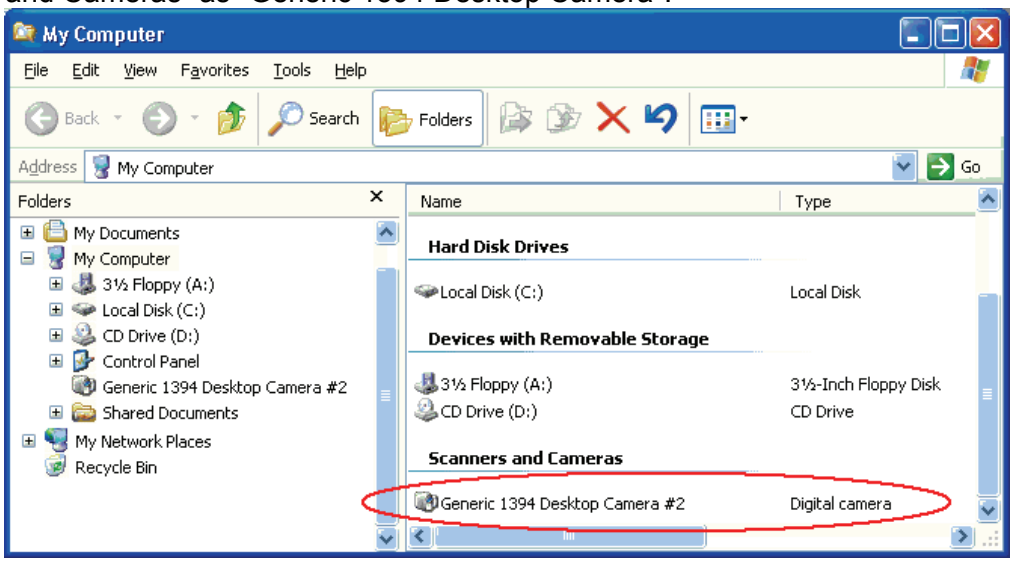

COM4_115200_8_N_1_HW - HyperTerminal
Datei Bearbeiten Ansicht Anrufen Übertragung ?
C6412Compact Video Processing Example Vr. 1.0 Oct 15 2008 11:49:56
Connect camera to IEEE1394 interface #1 (connector next to ethernet connector)
Connect host (PC) to IEEE1394 interface #2 (connector next to EGND connector)
Note: the camera must support 640x480 YUV4:2:2 30 fps
IIDC 1394 Camera video registers can be accessed now on output bus
IIDC 1394 Camera feature registers can be accessed now on output bus
Bus reset on input side.
Connecting to: Unibrain Fire-I 400 1.2 S/W v00000100h
Frame Size 153600 Byte
Packet Size 640 Byte
Packets Per Frame 240
Bus reset on output side.
Verbunden 02:56:41
ANSI
115200 8-N-1
RF
GROSS
NUM
Aufzeichnen
Druckerecho

[2] If no terminal program is available or message output is not desired, the RS-232 interface of the C6412Compact must have a handshake loopback with pins 7 and 8 connected together.

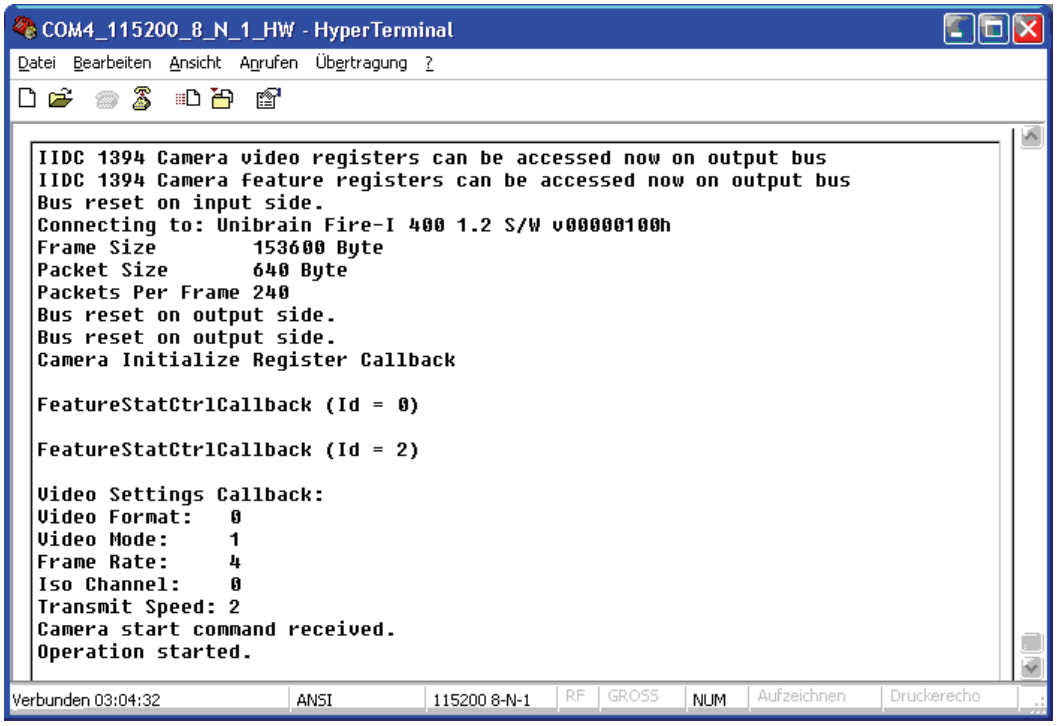
- The PC recognizes the C6412Compact Video Processing Development Kit as an IIDC 1394 compliant camera, loads a high-level driver for the IIDC device class and then initializes the camera. This is indicated by the last two lines in the figure below:


COM4_115200_8_N_1_HW - HyperTerminal
Datei Bearbeiten Ansicht Anrufen Übertragung ?
C6412Compact Video Processing Example Vr. 1.0 Oct 15 2008 11:49:56
Connect camera to IEEE1394 interface #1 (connector next to ethernet connector)
Connect host (PC) to IEEE1394 interface #2 (connector next to EGND connector)
Note: the camera must support 640x480 YUV4:2:2 30 fps
IIDC 1394 Camera video registers can be accessed now on output bus
IIDC 1394 Camera feature registers can be accessed now on output bus
Bus reset on input side.
Connecting to: Unibrain Fire-I 400 1.2 S/W v00000100h
Frame Size 153600 Byte
Packet Size 640 Byte
Packets Per Frame 240
Bus reset on output side.
Bus reset on output side.
Camera Initialize Register Callback
Verbunden 02:56:41
ANSI
115200 8-N-1
RF
GROSS
NUM
Aufzeichnen
Druckerecho

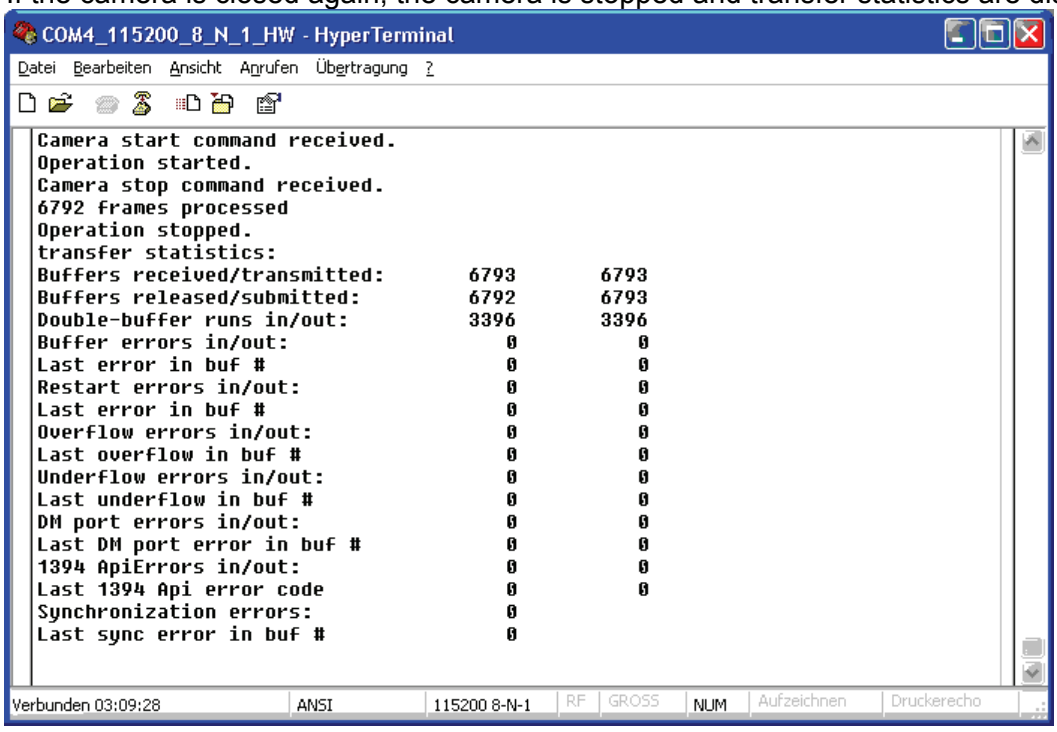
- On the development PC, click on "Start", then on "My Computer"
- The DCAM Video Processing Development Kit must now be displayed under "Scanners and Cameras" as "Generic 1394 Desktop Camera":



- If the camera is opened by a double-click, the PC sets up and starts the camera and displays the received images. The application example on the C6412Compact sets up and starts the digital camera connected to IEEE1394 interface 1. The incoming image data is processed and the results are sent to the PC connected to IEEE1394 interface 2. The red led of the C6412Compact indicates image processing activity. The resulting message output is shown below:


COM4_115200_8_N_1_HW - HyperTerminal
Datei Bearbeiten Ansicht Anrufen Übertragung ?
IIDC 1394 Camera video registers can be accessed now on output bus
IIDC 1394 Camera feature registers can be accessed now on output bus
Bus reset on input side.
Connecting to: Unibrain Fire-I 400 1.2 S/W v00000100h
Frame Size 153600 Byte
Packet Size 640 Byte
Packets Per Frame 240
Bus reset on output side.
Bus reset on output side.
Camera Initialize Register Callback
FeatureStatCtrlCallback (Id = 0)
FeatureStatCtrlCallback (Id = 2)
Video Settings Callback:
Video Format: 0
Video Mode: 1
Frame Rate: 4
Iso Channel: 0
Transmit Speed: 2
Camera start command received.
Operation started.
Verbunden 03:04:32 ANSI 115200 8-N-1 RF GROSS NUM Aufzeichnen Druckerecho

- If the camera is closed again, the camera is stopped and transfer statistics are displayed:


COM4_115200_8_N_1_HW - HyperTerminal
Datei Bearbeiten Ansicht Anrufen Übertragung ?
Camera start command received.
Operation started.
Camera stop command received.
6792 frames processed
Operation stopped.
transfer statistics:
Buffers received/transmitted: 6793 6793
Buffers released/submitted: 6792 6793
Double-buffer runs in/out: 3396 3396
Buffer errors in/out: 0 0
Last error in buf # 0 0
Restart errors in/out: 0 0
Last error in buf # 0 0
Overflow errors in/out: 0 0
Last overflow in buf # 0 0
Underflow errors in/out: 0 0
Last underflow in buf # 0 0
DM port errors in/out: 0 0
Last DM port error in buf # 0 0
1394 ApiErrors in/out: 0 0
Last 1394 Api error code 0 0
Synchronization errors: 0
Last sync error in buf # 0
Verbunden 03:09:28 ANSI 115200 8-N-1 RF GROSS NUM Aufzeichnen Druckerecho

The application example is described in detail in chapter 5.

4 Software Development

This chapter describes what is necessary to develop video processing applications on the C6412Compact. Further hints for application development can be found throughout chapter 5.1.

4.1 Installation

Simply copy the contents of the distribution media to a convenient location on your development PC, e.g. *c:\Orsys*. Then add the path to the Flash File System utilities to the *Path* environment variable, e.g. `set PATH=%PATH%:c:\Orsys\PC_FlashFileSystem_Tools`. For a description of how to set environment variables, please refer to your operating system documentation.

4.2 Required Tools

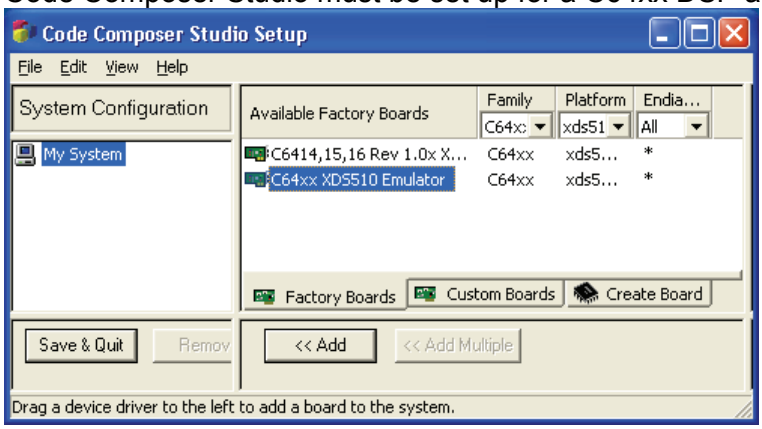
For program development, debugging and download you need

- a development PC
- a JTAG emulator, from Texas Instruments (e.g. XDS510) or from another vendor
- Code Composer Studio (CCS) from Texas Instruments, version 3.3 or higher.
- DSP/BIOS from Texas Instruments, version 5.31.02 or higher

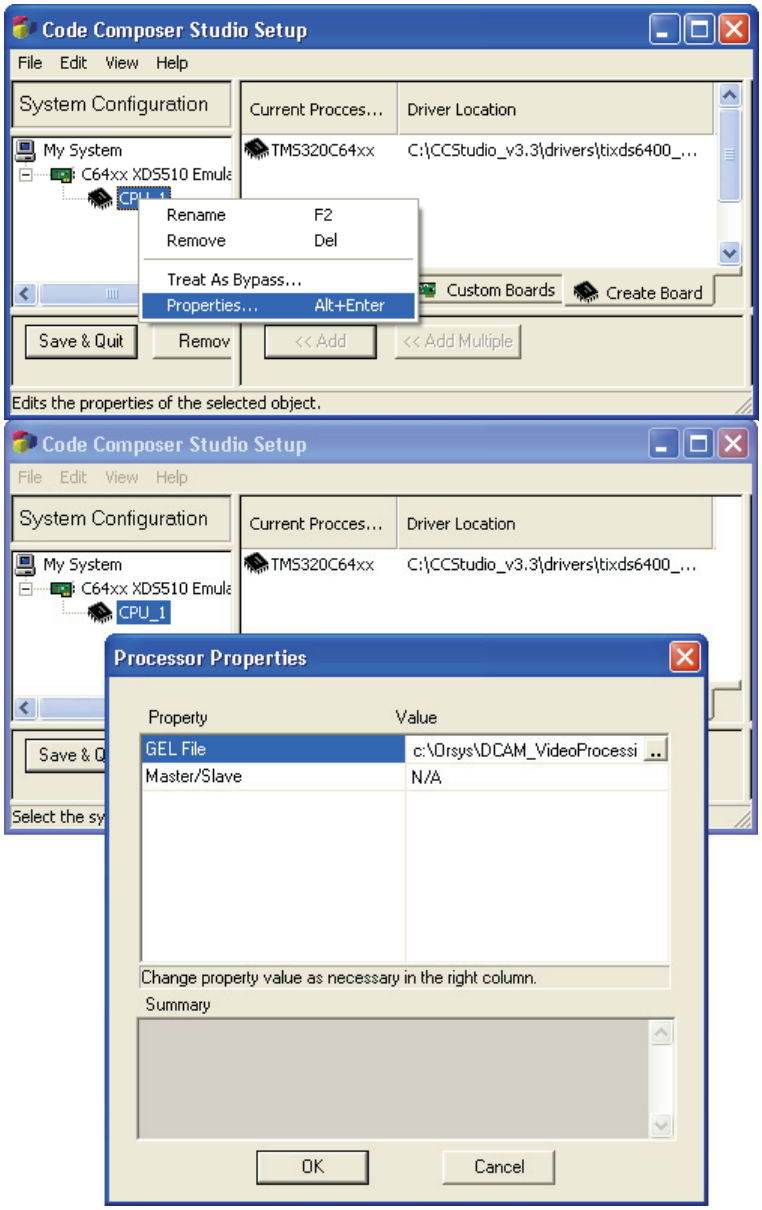
Other development platforms may be supported by Texas Instruments or other vendors. Please refer to [1] for a list of supported platforms and third party products. Development without a JTAG emulator is possible in principle, but not recommended since no debugging features are available in this configuration.

4.2.1 Code Composer Studio Setup

Code Composer Studio must be set up for a C64xx DSP and the respective emulator type.



Additionally it is useful to replace the default GEL file by an appropriate GEL file from the distribution copy, such as `c6412cpt_ml_bm_128mb.gel`. This can be done by right-clicking onto the CPU symbol and then selecting "Properties":



4.3 Development Flow

Development flow is the same as described in [11] for DSP/BIOS applications, with the following additional requirements:

- The necessary header files for the DCAM libraries must be included where required in the source code. These are combined into one header file, *DcamVideoAPI.h*.
- The chip support library (*csl6412.lib*) must be added to the project (as source file or as linker option)
- The DCAM libraries (*dcam_frame_capture.lib*, *dcam_camera.lib*) must be added to the project (as source file or as linker option)
- The IEEE1394 libraries (*lynxapi1.lib*, *lynxapi2.lib*, *lynxhal1.lib*, *lynxhal2.lib*) must be added to the project (as source file or as linker option)
- The software streaming library (*sw_str.lib*) must be added to the project (as source file or as linker option)
- The board library (*c6412cpt.lib*) must be added to the project (as source file or as linker option)

4.4 Required DSP/BIOS Settings and Objects

Creating a video processing application requires a number of settings and DSP/BIOS objects to be defined in a DSP/BIOS configuration script (a file with `.tcf` suffix). DSP/BIOS configuration files can be created with Code Composer Studio using "File"→"New" "DSP/BIOS configuration…". However, in most cases it is more effective to use the configuration file of the application example (*VP_example_720MHz_64MB.tcf*) as a template and to modify it. Most settings can be modified by the DSP/BIOS Configuration Tool as described in [11], whereas some settings require editing the configuration file with a text editor where noted.

4.4.1 Memory Layout

The application example divides the available SDRAM into two halves for usage by application code and as frame buffers:

```
/* loading the generic platform */
var mem_ext = [];
mem_ext[0] = {
    comment: "on-board SDRAM (lower 32MB)",
    name: "ERAM_APP",
    base: 0x80000000,
    len:  0x02000000,
    space: "code/data"
};
mem_ext[1] = {
    comment: "on-board SDRAM (upper 32MB)",
    name: "ERAM_BUFFERS",
    base: 0x82000000,
    len:  0x02000000,
    space: "code/data"
};
```

This was done to have a strict separation of application and frame buffers. Future version may require a similar memory layout to support PDT transfers, but currently this is not supported. If the two-section memory layout is not desired, the default memory layout can be taken from the board library's configuration files, e.g. *c6412cpt_720MHz_64MB.tcf*. This must be done manually with a text editor.

```
/* loading the generic platform */
var mem_ext = [];
mem_ext[0] = {
    comment: "on-board SDRAM (64MB)",
    name: "ERAM",
    base: 0x80000000,
    len:  0x04000000,
    space: "code/data"
};
```

VP_example was configured for a C6412Compact with 64MB SDRAM. To modify (the two-section memory layout) for 128MB, the following changes must be done with a text editor:

```
/* loading the generic platform */
var mem_ext = [];
mem_ext[0] = {
    comment: "on-board SDRAM (lower 64MB)",
    name: "ERAM_APP",
    base: 0x80000000,
    len:  0x04000000,
    space: "code/data"
};
mem_ext[1] = {
    comment: "on-board SDRAM (upper 64MB)",
    name: "ERAM_BUFFERS",
    base: 0x84000000,
    len:  0x04000000,
    space: "code/data"
};
...
bios.GBL.C641XMARCE0 = 0x00ff;
```

4.4.2 Clock Rate

By default, the application example is set up for a clock rate of 720 MHz, which is recommended for video processing operation. If a C6412Compact with 600 MHz is used, the configuration file must be modified as follows with a text editor:

```
/* The structure 'params' is described in detail in the Tconf User's Guide,
 * chapter "Tconf Platform Files". The Guide (SPRU007G) is located in
 * the xdctools/doc directory.
 */
var params = {
    clockRate: 600,
    catalogName: "ti.catalog.c6000",
    deviceName: "6412",
    regs: device_regs,
    mem: mem_ext
};
```

4.4.3 Cache Setup

By default the application example uses the entire DSP on-chip memory as L2 cache:

```
var device_regs = {
    /* The property l2Mode specifies the L2 cache setting. The valid values are:
     * "4-way cache (0k)", "4-way cache (32k)", "4-way cache (64k)",
     * "4-way cache (128k)", "4-way cache (256k)"
     */
    l2Mode: "4-way cache (256k)"
};
```

For some applications it may be better to have parts of code and/or data in DSP on-chip memory and to reduce the size of L2 cache accordingly. This can be done with a text editor as shown below.

```
/* loading the generic platform */
var mem_ext = [];
mem_ext[0] = {
    comment: "DSP on-chip memory",
    name: "IRAM",
    base: 0x00000000,
    len:  0x00040000,
    space: "code/data"
};
mem_ext[1] = {
    comment: "on-board SDRAM (lower 32MB)",
    name: "ERAM_APP",
    base: 0x80000000,
    len:  0x02000000,
    space: "code/data"
};
mem_ext[2] = {
    comment: "on-board SDRAM (upper 32MB)",
    name: "ERAM_BUFFERS",
    base: 0x82000000,
    len:  0x02000000,
    space: "code/data"
};

…

var device_regs = {
    /* The property l2Mode specifies the L2 cache setting. The valid values are:
     * "4-way cache (0k)", "4-way cache (32k)", "4-way cache (64k)",
     * "4-way cache (128k)", "4-way cache (256k)"
     */
    l2Mode: "4-way cache (128k)"
};
```

Completely disabling the cache is not recommended, as this probably slows down processing and may interfere with the cache management of the software streaming library.

4.4.4 Heap Setup

The application example uses a heap size of 640KB:

```
bios.MEM.instance("ERAM_APP").createHeap = 1;
bios.MEM.instance("ERAM_APP").heapSize = 0x000a0000;
```

The required heap size strongly depends on the application. When opening the DSP/BIOS configuration file, the estimated heap size is displayed in the title bar.

4.4.5 Section location

The default section for DSP/BIOS objects and heap objects created by *malloc()* must be changed:

```
bios.MEM.BIOSOBJSEG = prog.get("ERAM_APP");
```

```
bios.MEM.MALLOCSEG = prog.get("ERAM_APP");
```
This is only necessary if the modified memory layout as described above is used.

4.4.6 Interrupts

Interrupt polarity is set to low-active / falling-edge triggered, as also used in the board library's settings. This setting must not be changed.

```
bios.HWI.EXTPIN4POLARITY = "high-to-low";
bios.HWI.EXTPIN5POLARITY = "high-to-low";
bios.HWI.EXTPIN6POLARITY = "high-to-low";
bios.HWI.EXTPIN7POLARITY = "high-to-low";
```

Each of the DCAM APIs required a hardware interrupt for operation. Additionally, the software streaming library requires the chip support library's (CSL's) EDMA interrupt handler to be installed. The API ISRs must run without the DSP/BIOS interrupt scheduler, whereas the EDMA interrupt is designed for scheduler operation.

```
bios.HWI.instance("HWI_INT4").fxn = prog.extern("LynxHALNodeISR1");
bios.HWI.instance("HWI_INT5").fxn = prog.extern("LynxHALNodeISR2");
bios.HWI.instance("HWI_INT8").fxn = prog.extern("EDMA_intDispatcher");
bios.HWI.instance("HWI_INT8").useDispatcher = 1;
```

4.4.7 Tasks

The task scheduler must be enabled:

```
bios.TSK.ENABLETSK = 1;
```

Three tasks are required by *VP_example*. Two of these tasks are application specific (*TSK_main* and *TSK_frame_xfer*) and can be modified freely, but one task (*TSK_acr_timeout*) is required by the DCAM Camera API for advanced feature timeout control:

```
bios.TSK.create("TSK_main");
bios.TSK.instance("TSK_main").order = 1;
bios.TSK.instance("TSK_main").fxn = prog.extern("FXN_main");
bios.TSK.instance("TSK_main").comment = "main control task";
bios.TSK.instance("TSK_main").stackSize = 2048;
bios.TSK.create("TSK_frame_xfer");
bios.TSK.instance("TSK_frame_xfer").order = 2;
bios.TSK.instance("TSK_frame_xfer").comment = "frame buffer management and image processing";
bios.TSK.instance("TSK_frame_xfer").priority = -1;
bios.TSK.instance("TSK_frame_xfer").fxn = prog.extern("FXN_frame_xfer");
bios.TSK.create("TSK_AcrTimeout");
bios.TSK.instance("TSK_AcrTimeout").order = 3;
bios.TSK.instance("TSK_AcrTimeout").fxn = prog.extern("FXN_AcrTimeout");
bios.TSK.instance("TSK_AcrTimeout").comment = "Timeout for advanced feature access control register
(ACR)";
```

4.4.8 Semaphores

For interprocess communication, *VP_example* uses three semaphores. Two of them synchronize between the software streaming library and the application's tasks. *SEM_FrameRx* is posted each time an incoming frame was completely received. This semaphore wakes up the image processing task. As a second condition for image processing to start, *SEM_FrameTx* is posted whenever an outgoing frame has been completely sent. A third semaphore, *SEM_acr_timeout*, is used internally by the DCAM Camera API for handling advanced feature access timeouts. For

```
bios.SEM.create("SEM_FrameRx");
bios.SEM.instance("SEM_FrameRx").comment = "Triggers image processing whenever a new frame was
received";
bios.SEM.create("SEM_FrameTx");
bios.SEM.instance("SEM_FrameTx").comment = "Triggered whenever a frame has been sent";
bios.SEM.create("SEM_AcrTimeout");
bios.SEM.instance("SEM_AcrTimeout").comment = "Timeout for advanced feature access control register
(ACR)";
```

Video processing applications should use *SEM_FrameRx* for synchronization, whereas usage of *SEM_FrameTx* is optional. Name of the semaphores can be changed, but must correspond with the object names used in the application's source code. Usage of *SEM_acr_timeout* is mandatory.

4.4.9 Resource Locks

The software streaming library uses two lock objects to lock against multiple accesses to the both IEEE1394 interfaces. These lock handles must be used whenever the software streaming library is used for isochronous data transfer.

```
bios.LCK.create("LCK_In");
bios.LCK.instance("LCK_In").comment = "Lock handle for IEEE1394 interface (input side)";
bios.LCK.create("LCK_Out");
bios.LCK.instance("LCK_Out").comment = "Lock handle for IEEE1394 interface (output side)";
```

4.4.10 Frame Buffers

The application implements its frame buffers as buffer objects with separate buffers for input and output and two buffers per direction:

```
bios.BUF.create("BUF_In");
bios.BUF.instance("BUF_In").align = 8;
bios.BUF.instance("BUF_In").size = 153600;
bios.BUF.instance("BUF_In").comment = "Input frame buffers";
bios.BUF.instance("BUF_In").bufSeg = prog.get("ERAM_BUFFERS");
bios.BUF.instance("BUF_In").bufCount = 2;
bios.BUF.create("BUF_Out");
bios.BUF.instance("BUF_Out").align = 8;
bios.BUF.instance("BUF_Out").size = 153600;
bios.BUF.instance("BUF_Out").comment = "Output frame buffers";
bios.BUF.instance("BUF_Out").bufSeg = prog.get("ERAM_BUFFERS");
bios.BUF.instance("BUF_Out").bufCount = 2;
```

This buffer scheme is not mandatory and can be changed freely. Examples for possible implementations are:

- No buffer objects, manual calculation of frame buffer address and size
- Single-section memory layout (ERAM_APP and ERAM_BUFFERS replaced by ERAM, as implemented with the board library's configuration files) with buffer objects placed in same memory section as application code / data.
- Common buffers for both, input and output (in-place image processing), only one buffer object needed. This is recommended because it can reduce the number of SDRAM accesses by up to 50% if the cache can buffer the accesses. However, many image processing algorithms do not allow in-place processing. Examples for in-place image processing are histogram calculation or overlaying fixed images into the camera's image.

If multiple video settings must be supported, the size of a buffer must be set to the maximum possible frame size and the buffer start addresses for smaller frame size must be calculated manually.

4.4.11 Idle Tasks

VP_example uses a background task for debug message output. This background task is a function from the board library that transmits one character to the UART per call if possible. Applications that do not need message output can omit this part.

```
bios.IDL.create("IDL_DebugOut");
bios.IDL.instance("IDL_DebugOut").order = 1;
bios.IDL.instance("IDL_DebugOut").comment = "Debug messages character output ";
bios.IDL.instance("IDL_DebugOut").fxn = prog.extern("StdioifDevWrite");
```

4.5 Loading Software Using a JTAG Emulator

These are the required steps for loading application example *VP_example* to the C6412Compact:

- CCS must be already installed.
- Start a terminal program, such as “HyperTerminal” on the PC.
- Set up the terminal program to 115200 bits per second, 8 data bits, no parity, 1 stop bit and hardware flow control.
- Connect the kit to the PC using the RS-232 cable.
- Connect the digital camera to IEEE1394 interface 1 of the C6412Compact using an IEEE1394 cable.
- Connect IEEE1394 interface 2 of the C6412Compact to the PC using an IEEE1394 cable.

- Connect your C6412Compact with your development PC by using the JTAG emulator and the JTAG adapter, which is delivered together with the kit.
- Supply the kit with power.
- Start CCS.
- Select "Load Gel..." from the "File" menu. The "*Load GEL File*" dialog box appears. Load the GEL file from the *C6412Compact\gel* folder within your distribution copy. Click on "Open" to load this GEL file.
- Select the "Reset and Init" command from the "GEL" → "Reset" menu. This will reset the DSP and set up the EMIF correctly, so that the DSP is ready to start. This also clears interrupts that may be still pending from previous debug sessions. Therefore, it is a good practice to perform this step each time before a program is loaded.
- Select the "Open..." command from the "Project" menu.
- The "*Project Open*" dialog box appears. Locate the file *VP_example.pjt* in the *examples\VP_example* folder of your distribution copy and click on the Open button. This loads the project environment for the application example.
- Select the "Load Program..." command from the "File" menu.
- The "*Load Program*" dialog box appears. Locate the file *camera.out* in the folder *C6412Compact\examples\VP_example\Debug* and click on the "Open" button. This loads the application example into the DSP's memory.
- Select the "Run" command from the "Debug" menu.
- *VP_example* starts and searches for a digital camera connected to the IEEE1394 interface 1.
- The PC recognizes the kit as an IIDC 1394 compliant camera.
- Click on "Start", then on "My Computer".
- The DCAM Video Processing Development Kit must now be displayed under "Scanners and Cameras" as "Generic 1394 Desktop Camera".

4.6 Debugging Software Using a JTAG Emulator

Select the "Halt" command from the "Debug" menu. CCS will stop the DSP and display the current instruction in the disassembly window and the current source code line (if applicable) in a separate window. Now you can examine variables (e.g. using the watch window) and memory contents ("View" → "Memory") and you can single-step through the code using F10 and F11. This is the traditional debugging approach that halts program execution. For DSP/BIOS applications a different approach is used: The DSP/BIOS analysis tools allow real-time analysis of a running application, such as program tracing, performance monitoring and file streaming. Please refer to [11] for details.

4.7 Storing Application Software in Flash Memory

Loading software with the emulator as described above is only possible if development tools are available. Further, the software must be re-loaded after each system startup. The steps below show how to store the example to the FFS, so that is automatically started at system startup.

- Close or disconnect the terminal program if still running to make the COM port available to the FFS utilities.
- If CCS is still connected to the C6412Compact ("Connected" displayed in the status bar), disconnect by selecting "Disconnect" from the "Debug" menu or by pressing Alt + C.
- Press the reset button on the power supply board.
- Start a command prompt
- Change to the directory of one of the example applications, e.g. `cd c:\Orsys\DCAM_VP_DevKit_v1.0\C6412Compact\examples\VP_example`
- Store the application code to flash memory using the *fload6x* utility. For the first time it may be necessary to specify correct parameters, such as COM port number, board type and version of the target-side executable, so the command line will look as follows:

`fload6x -C1 -TC6412CPT -V4.0 vp_example`. The settings above are saved; therefore for subsequent use of this utility a simpler command line can be used: `fload6x vp_example`. For detailed information about *fload6x* and its parameters, please refer to [2].

- If *fload6x* is continuously displaying asterisks ('*'), press the reset button on the power supply board. In some rare cases, if there is already an application in flash memory, it may be necessary to hold the reset button, start *fload6x* and then release the reset button.
- *fload6x* first loads a target executable to the C6412Compact and then the application code, which is programmed to flash memory. After execution has finished, the C6412Compact is automatically reset, starts up and loads *vp_example*. If the terminal is started (or connected) again, the startup message is displayed and the example can be used.

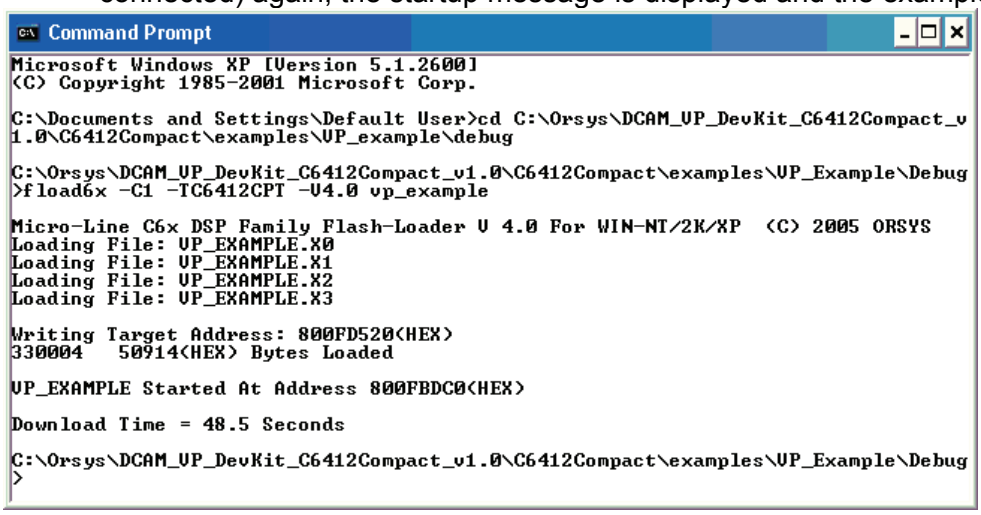

Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Default User>cd C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\examples\VP_example\debug
C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\examples\VP_Example\Debug>fload6x -C1 -TC6412CPT -V4.0 vp_example
Micro-Line C6x DSP Family Flash-Loader V 4.0 For WIN-NT/2K/XP (C) 2005 ORSYS
Loading File: VP_EXAMPLE.X0
Loading File: VP_EXAMPLE.X1
Loading File: VP_EXAMPLE.X2
Loading File: VP_EXAMPLE.X3
Writing Target Address: 800FD520(HEX)
330004 50914(HEX) Bytes Loaded
VP_EXAMPLE Started At Address 800FBDC0(HEX)
Download Time = 48.5 Seconds
C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\examples\VP_Example\Debug>

Figure 3: Session example: storing application code to flash memory

A detailed description of all Flash File System commands can be found in [2].

4.8 Storing FPGA Code in Flash Memory

As already mentioned in section 2.2.6, the C6412Compact already has FPGA code stored in the FFS, so that it can instantly be used. However, storing FPGA code to the FFS is necessary

- When FPGA code updates must be installed.
- If a customized FPGA design is used.
- If FPGA code was accidentally deleted.
- For deployment of end systems. Here, the C6412Compact is purchased standalone and has only the default application (*toggle_led*) in the FFS.

The steps below show how to store the micro-line® busmaster FPGA to flash memory for a 4Mgates (XC3S4000) device:

- Close or disconnect the terminal program if still running to make the COM port available to the FFS utilities.
- If CCS is still connected to the C6412Compact ("Connected" displayed in the status bar), disconnect by selecting "Disconnect" from the "Debug" menu or by pressing Alt + C.
- Press the reset button on the power supply board.
- Start a command prompt.
- Change to the folder where the FPGA code for the micro-line® busmaster BSP resides, e.g. *c:\Orsys\DCAM_VP_DevKit_v1.0\C6412Compact\fpga_binaries\ML_Busmaster_v2.1*.
- Store the FPGA code to flash memory using the *fpgaload* utility. To store the FPGA code, but preventing auto-configuration at startup, option *–N0* must be specified:

  `fpgaload -N0 ml_bm_4m`.

  If auto-configuration is desired, *–N1* must be specified or the *–N* option must be omitted:

  `fpgaload ml_bm_4m`.

For detailed information about *fpgaload* and its parameters, please refer to [2].

- If *fpgaload* is continuously displaying asterisks ('*'), press the reset button on the power supply board. In some rare cases, if there is already an application in flash memory, it may be necessary to hold the reset button, start *fpgaload* and then release the reset button.
- *fpgaload* first loads a target executable to the C6412Compact and then the FPGA code, which is programmed to flash memory. After execution has finished, the C6412Compact is automatically reset, starts up and (if *–N0* was not specified) configures the FPGA.
- Please note that the application examples use a default name of *ML_BM* when accessing the FFS for FPGA code. Therefore, the file just uploaded must be renamed as follows:

```
renml ml_bm_4m ml_bm.
```

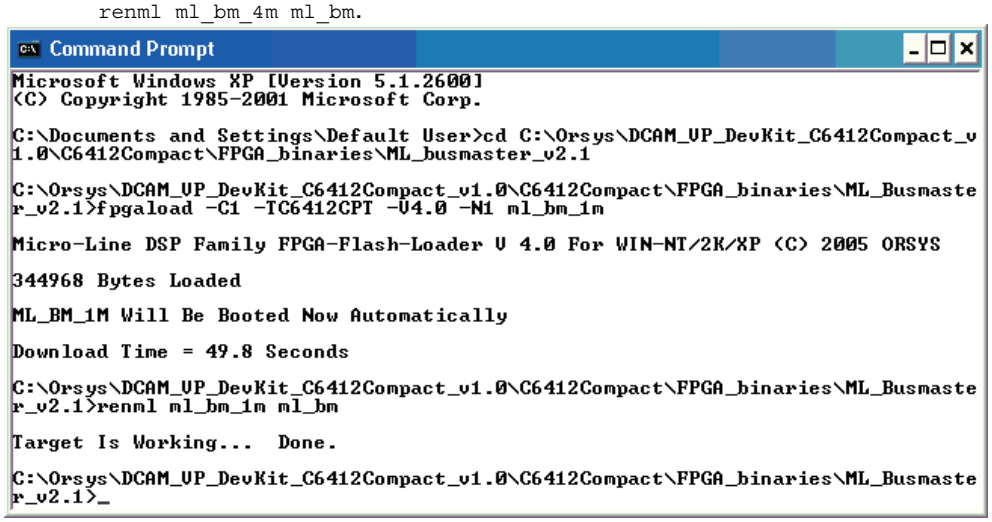

Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Default User>cd C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\FPGA_binaries\ML_busmaster_v2.1
C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\FPGA_binaries\ML_Busmaster_v2.1>fpgaload -C1 -TC6412CPT -V4.0 -N1 ml_bm_1m
Micro-Line DSP Family FPGA-Flash-Loader V 4.0 For WIN-NT/2K/XP (C) 2005 ORSYS
344968 Bytes Loaded
ML_BM_1M Will Be Booted Now Automatically
Download Time = 49.8 Seconds
C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\FPGA_binaries\ML_Busmaster_v2.1>renml ml_bm_1m ml_bm
Target Is Working... Done.
C:\Orsys\DCAM_VP_DevKit_C6412Compact_v1.0\C6412Compact\FPGA_binaries\ML_Busmaster_v2.1>_

Figure 4: Session example: storing FPGA code to flash memory

5 Application Examples

The distribution media contains two kinds of application examples:

- One application example that shows how to program a video processing application using the DCAM Frame Capture API and the DCAM Camera API
- Some examples that are part of existing software packages, such as the board library. These examples show general programming of the C6412Compact, independent of video processing. They are described in the respective user guides, such as [8].

This chapter describes the video processing application example (named *VP_example*) in detail.

5.1 VP_example

VP_example implements a simple processing algorithm: The incoming picture is dimmed by 50% intensity, except for those pixels that have changed with respect to the last frame. This has the effect of highlighting changes in the image.

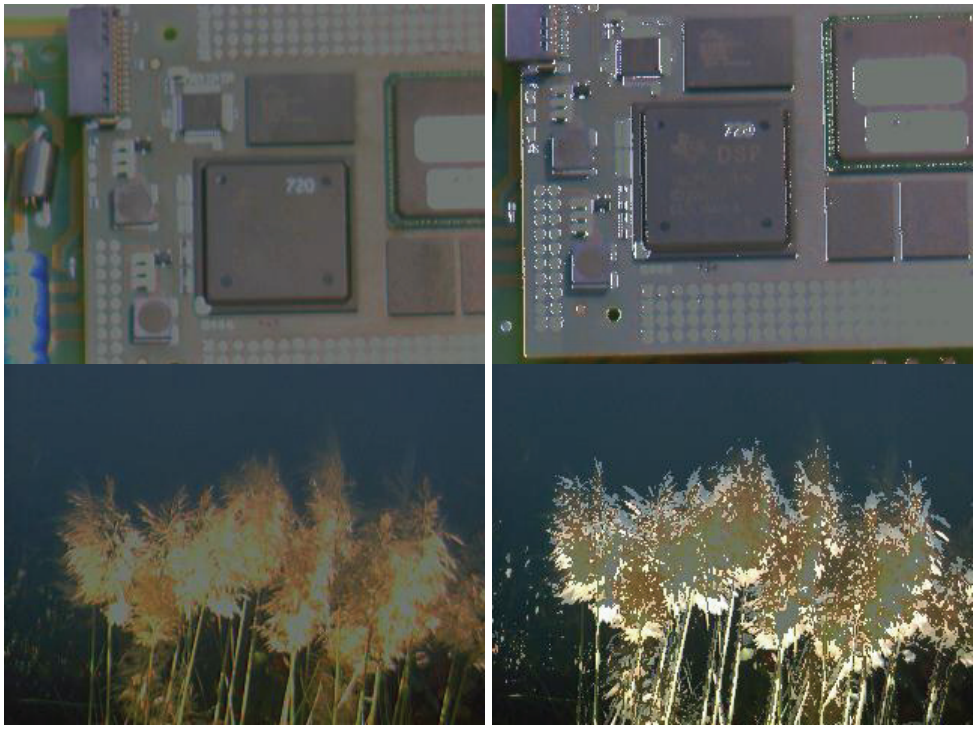


Figure 5: VP_example output (still image vs. moving)

The example is written as a DSP/BIOS application, configured for 720MHz CPU clock and 64MB SDRAM. If a C6412Compact with different settings is used, a different DSP/BIOS configuration script must be used. It can be created from the default script, *VP_example_720MHz_64MB.tcf*. Note: as indicated later on, the example uses GP0[11] and GP0[12] for triggering EDMA transfers. These signals are available on micro-line® connector pins B23 and B24 respectively for measurements. If the example is running on a different hardware environment, care must be taken that these pins are not connected otherwise.

5.1.1 Application structure

VP_example consists of 4 modules:

- *main.c* contains system initialization, the main control task and the frame processing task
- *img_proc.c* contains the image processing algorithm

- *camera_callbacks.c* handles incoming control requests from a camera controller (typically a PC)
- *camera_setup* initializes the output side with the desired features and settings

5.1.2 System initialization

System initialization is located in function main in module *main.c*. This includes all steps that do not require DSP/BIOS. First step is to initialize the chip support library. This must be done by application code starting with DSP/BIOS v5.x.

```
void main (void)
{
    INT32U      ulTmp;
    DCAM_STATUS Status;    /* return status of DCAM API calls */

    /* initialize CSL */
    CSL_init();
```

After CSL initialization, interrupts are set up. Interrupts from the FPGA are mapped to CPU interrupts, all interrupts used by this example are cleared and finally interrupts are enabled.

```
    /* map LLC interrupts to DSP interrupts in PLD for operation of both 1394 APIs */
    C6412CPT_INTCFG1 = (C6412CPT_INTCFG1_LLC1_EXTINT4 |
                        C6412CPT_INTCFG1_LLC2_EXTINT5);

    /* clear interrupts used by this example. This may be useful if when */
    /* re-starting during a debug session */
    IRQ_clear(IRQ_EVT_EXTINT4);
    IRQ_clear(IRQ_EVT_EXTINT5);
    IRQ_clear(IRQ_EVT_EDMAINT);
    /* enable interrupts globally */
    IRQ_globalEnable();
    /* enable EDMA end-of-transfer interrupts */
    IRQ_enable(IRQ_EVT_EDMAINT);
```

Then the call queue for the IEEE1394 APIs is initialized. Currently, the API uses its own callback queue. This may be replaced by DSP/BIOS mechanisms in future versions.

```
    /* initialize callback queue for IEEE1394 API */
    InitCallQ();
```

Message output is done over RS-232, using the stream interface of the runtime library. Below is the code for setting up and doing stream-based output.

```
    // initialize the stdio interface to use on-board UART
    StdioifInit ();

    // open Uart device for debug output and input
    // and configure for line-buffered output with default buffer size
    fid_write = fopen ("UART:DummyName", "w");
    fid_read  = fopen ("UART:DummyName", "r");
    setvbuf(fid_write, NULL, _IOLBF, BUFSIZ);

    // display startup message
    fprintf (fid_write, "\r\n\r\n%s\r\n", acVersion);
    fprintf (fid_write, "Connect camera to IEEE1394 interface #1
             "(connector next to ethernet connector)\r\n");
    fprintf (fid_write, "Connect host (PC) to IEEE1394 interface #2 "
             "(connector next to EGND connector)\r\n");
    fprintf (fid_write, "Note: the camera must support 640x480 YUV4:2:2 30 fps\r\n");
    StdioifFlush (); //only required/allowed in main()
```

The application example uses the micro-line busmaster BSP for IEEE1394 operation. Therefore the FPGA must be loaded with the appropriate code.

```
    /* Make sure the FPGA is configured correctly. We need ML busmaster type, v2.x or higher */
    if ((((C6412CPT_FPGAVER & C6412CPT_FPGAVER_TYPE_MASK) != C6412CPT_FPGAVER_TYPE_ML_BM ||
        ((C6412CPT_FPGAVER & C6412CPT_FPGAVER_VER_MASK) < (2 << C6412CPT_FPGAVER_VER_SHIFT))
    {
        /* try to configure the FPGA from flash memory */
        if (FpgaLoadOnboardFromFlash("ML_BM", NULL) != FPGA_OK)
        {
            /* no FPGA available; abort */
            fprintf (fid_write, "No suitable FPGA available. "
                     "Need ML busmaster v2.0 or higher\r\n");
            StdioifFlush (); //only required/allowed in main()
            C6412CPT_RED_LED_ON;
            SYS_exit (1);
        }
```

```
}
```

Next step is initialization of the DCAM APIs (DCAM Frame Capture API for the input side and DCAM Camera API for the output side). Both APIs use IDs stored in the Flash File System for unique identification on the IEEE1394 interface. Initialization is done in two steps.

```
/* initialize 1394 interface #1 as input from a camera (DCAM frame */
/* capture side) */
sinCameraInit.pBusResetCompleteCallback = appinBusResetCallback;
sinCameraInit.pErrorDetectedCallback = NULL;
if (FfsDescrQueryDec("[Board]", "SerialNumber", &sinCameraInit.ulChipIdLow) != FFS_DESCR_OK ||
    FfsDescrQueryDec("[1394_1]", "ChipIdHigh",  &u32Tmp)                    != FFS_DESCR_OK)
{
    fprintf (fid_write, "Can't get 1394 IDs from FFS\r\n");
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
sinCameraInit.ucChipIdHigh = u32Tmp;
Status = dcaminInitialize (&sinCameraInit);
if (Status != DCAMIN_SUCCESS)
{   fprintf (fid_write, "dcaminInitialize failed with code %04Xh\r\n", Status);
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
Status = dcaminIsoInit();
if (Status != DCAMIN_SUCCESS)
{   fprintf (fid_write, "dcaminIsoInit failed with code %04Xh\r\n", Status);
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
/* initialize 1394 interface #2 as output to a camera controller / PC */
/* (DCAM camera side) */
soutCameraInit.pBusResetCompleteCallback = appoutBusResetCallback;
soutCameraInit.pErrorDetectedCallback    = NULL;
soutCameraInit.ulChipIdLow               = sinCameraInit.ulChipIdLow;
if (FfsDescrQueryDec("[1394_2]", "ChipIdHigh", &u32Tmp) != FFS_DESCR_OK)
{
    fprintf (fid_write, "Can't get 1394 IDs from FFS\r\n");
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
soutCameraInit.ucChipIdHigh = u32Tmp;
soutCameraInit.pcVendorName = acVendor;
soutCameraInit.pcModelName  = acModel;
Status = dcamoutInitialize (&soutCameraInit);
if (Status != DCAMOUT_SUCCESS)
{   fprintf (fid_write, "dcamoutInitialize failed with code %04Xh\r\n", Status);
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
Status = dcamoutIsoInit();
if (Status != DCAMOUT_SUCCESS)
{
    fprintf (fid_write, "dcamoutIsoInit failed with code %04Xh\r\n", Status);
    StdioifFlush (); //only required/allowed in main()
    C6412CPT_RED_LED_ON;
    SYS_exit (1);
}
```

After initialization of the APIs, the registers and callback functions for the output side are set up. This step requires a number of function calls and is therefore located in a separate module, *camera_setup.c*.

```
/* initialize output side. */
/* This step sets up the registers of the output side according to */
/* the IIDC spcification & implemented features. This involves */
/* - preparing initial register contents */
/* - mapping the registers into the IEEE1394 address space */
/* - installing callback functions for register access */
SetupDigitalCamera();
```

After initialization, API operation is started by enabling the interrupts and triggering a bus reset on both IEEE1394 interfaces. On the input side, the bus reset causes the bus scanned for connected cameras. On the output side, the bus reset tells a connected camera controller that bus topology has changed and that it should check for new devices on the bus.

```
/* prepare interrupt operation of both 1394 APIs */
IRQ_enable(IRQ_EVT_EXTINT4);
IRQ_enable(IRQ_EVT_EXTINT5);

/* trigger inital startup on both 1394 APIs */
dcaminCauseBusReset();
binBusResetDetected = TRUE; /* force bus reset processing in control task */

dcamoutCauseBusReset();
boutBusResetDetected = TRUE;/* force bus reset processing in control task */

/* end of system initialization. DSP/BIOS will start up now and then */
/* execute the main control thread, FXN_main. */
```

Now initialization is complete and control is passed to DSP/BIOS, which will run the different tasks that are listed below.

5.1.3 Main Control Task

The main control task controls operation of both sides. It is the only location where IEEE1394 transactions (except for the image transfer itself) are generated. No other tasks are allowed to do IEEE1394 transactions because the APIs are not reentrant.
The task is named *FXN_main* and is implemented as an infinite loop. If there is a camera available and there is no bus reset on either side, normal image processing is running and the task checks for start / stop / control events and processes them.

```
while(1)
{
    /*********************************************************************/
    while (binBusResetDetected  == FALSE &&
           boutBusResetDetected == FALSE &&
           hCamera              != NULL)
    {
        /* normal operation (suitable camera connected and no bus reset */
        /* present) */
        /* handle incoming commands (from ext. camera controller/PC */
        /* connected to the output side) */
        /*********************************************************************/
        if (bStartEvent == TRUE)
        {
            ...
        }
        /*********************************************************************/
        if (bStopEvent == TRUE)
        {
            ...
        }
        /*********************************************************************/
        if (bControlEvent == TRUE)
        {
            ...
        }
```

After processing these application-level events, API callbacks are processed and control is passed to DSP/BIOS to allow background tasks to run.

```
        ...
        /*********************************************************************/
        /* process IEEE1394 API callbacks */
        YieldToCallQ();
        /*********************************************************************/
        /* Allow background operation */
        TSK_sleep(1 * CLK_countspms() / CLK_getprd());
    } /* normal operation */
```

If there is no camera connected, start / stop events are ignored. Instead, bus reset events are processed as follows:

```
    ...
    } /* normal operation */
    /*********************************************************************/
    if (bStartEvent == TRUE)
    {
        fprintf (fid_write, "Camera start command ignored because no camera connected\r\n");
        bStartEvent = FALSE;
    }
    /*********************************************************************/
    if (bStopEvent == TRUE)
    {
```

```
        fprintf (fid_write, "Camera stop command ignored because no camera connected\r\n");
        bStopEvent = FALSE;
    }
    /*****************************************************************/
    if (binBusResetDetected == TRUE)
    ...
```

For the input side, API callbacks are processed until bus reset processing has completed. Then, camera management is done. If a camera was previously present, the software checks that the camera is still present. If the previous camera was removed or there was no previous camera, selection of a new camera is initiated. The first available camera is opened and checked to support the required settings. Then, the following settings are adjusted:

- Format 0 (VGA, non-compressed)
- Mode 1 (YUV 4:2:2, 16 bit per pixel)
- Rate 4 (30 frames per second)
- Isochronous transmit on channel 0
- Isochronous transmit speed S400 (400Mbit/s)

```
if (binBusResetDetected == TRUE)
{
    /* bus reset on input side: wait until bus reset processing is */
    /* over and then do camera connection management */
    fprintf (fid_write, "Bus reset on input side.\r\n");
    while(dcaminBusResetCompleted() == FALSE)
        YieldToCallQ();
    binBusResetDetected = FALSE;
    /* check if a camera is already selected */
    if (hCamera != NULL)
    {
        BOOL bTmp;
        /* check if this camera is still present */
        Status = dcaminIsCameraConnected (hCamera, &bTmp);
        if (Status != DCAMIN_SUCCESS || bTmp == FALSE)
        {
            fprintf (fid_write, "Connection to camera lost.\r\n");
            Status = dcaminIsoDeallocateResources(sCurIso.usIsoChannel);
            dcaminCloseCameraHandle (hCamera);
            hCamera = NULL;
        }
        else
        {
            fprintf (fid_write, "Camera still connected.\r\n");
        }
    }
    /* check if a camera must be selected */
    if (hCamera == NULL)
    {
        /* search for first camera, open it and set it up */
        iNumCameras = 1; /* do not find more than one */
        dcaminLocateCameras (&sCameraGuid, &iNumCameras);
        if (iNumCameras == 1)
        {
            /* open this camera */
            if ((Status =
                 dcaminOpenCameraHandle(&hCamera, &sCameraGuid))
                != DCAMIN_SUCCESS)
            {
                fprintf (fid_write, "can't open first camera: "
                         "%04Xh\r\n");
                hCamera = NULL;
            }
            else
            {
                fprintf (fid_write,
                         "Connecting to: %s %s S/W v%08Xh\r\n",
                         hCamera->csVendorName,
                         hCamera->csModelName,
                         hCamera->ulSoftwareVersion);
            }
        } /* open first camera */
        if (hCamera != NULL)
        {
            /* check if the desired format is available */
            DCAMINs_SUPPORTED_VIDEO sCaps;
            Status = dcaminQueryVideoCapabilities (hCamera, &sCaps);
            if (Status != DCAMIN_SUCCESS)
            {
                fprintf (fid_write, "Can't get supported "
                         "settings.\r\ndcaminQueryVideoCapabilities "
```

```
                        "returned %04Xh\r\n", Status);
                dcaminCloseCameraHandle(hCamera);
                hCamera = NULL;
            }
            else if ((sCaps.ulFormat           & 0x80000000) == 0 ||
                     (sCaps.sSupMode[0].ulMode & 0x40000000) == 0 ||
                     (sCaps.sSupMode[0].ulFrameRate[1]
                                               & 0x08000000) == 0)
            {
                fprintf (fid_write, "Camera doesn't supported desired "
                        "format/mode/rate.\r\n");
                dcaminCloseCameraHandle(hCamera);
                hCamera = NULL;
            }
        }
        if (hCamera != NULL)
        {
            /* set up format / mode / frame rate */
            // set video settings to 320x240 YUV(4:2:2), 30 fps
            sCurVideo.eVideoFormat = Format_0;  /* VGA */
            sCurVideo.eVideoMode   = Mode_1;    /* 320x240 YUV(4:2:2) */
            sCurVideo.eFrameRate   = FrameRate_4; /* 30 fps */
            Status = dcaminSetVideoData (hCamera, &sCurVideo);
            if (Status != DCAMIN_SUCCESS)
            {
                fprintf (fid_write, "Can't setup format/mode/rate: "
                        "dcaminSetVideoData returned %04Xh\r\n",
                        Status);
                dcaminCloseCameraHandle(hCamera);
                hCamera = NULL;
            }
        } /* format/mode/rate setup */
        if (hCamera != NULL)
        {
            /* set up isocohronous transmit settings */
            sCurIso.usIsoChannel   = 0;
            sCurIso.eTransmitSpeed = S400;
            Status = dcaminSetIsoData (hCamera, &sCurIso);
            if (Status != DCAMIN_SUCCESS)
            {
                fprintf (fid_write, "can't setup transmit parameters: "
                        "dcaminSetIsoData returned %04Xh\r\n", Status);
                dcaminCloseCameraHandle(hCamera);
                hCamera = NULL;
            }
            else
            {
                /* display resulting parameters */
                iFrameSize =
                    dcaminQueryFrameSize(sCurVideo.eVideoFormat,
                                         sCurVideo.eVideoMode);
                fprintf (fid_write, "Frame Size        %d Byte\r\n",
                        iFrameSize);
                iPacketSize =
                    dcaminQueryPacketSize (sCurVideo.eVideoFormat,
                                           sCurVideo.eVideoMode,
                                           sCurVideo.eFrameRate);
                fprintf (fid_write, "Packet Size       %d Byte\r\n",
                        iPacketSize);
                iPackPerFrame = iFrameSize / iPacketSize;
                fprintf (fid_write, "Packets Per Frame %d\r\n",
                        iPackPerFrame);
            }
        } /* isochronous parameters setup */
    } /* camera selection */
} /* input-side bus reset handling */
```

For the output side there is no special bus reset necessary. API callbacks are processed until bus reset processing has finished. This ensures that the output side is ready before a start command is being processed.

If there is neither a camera connected nor a bus reset in progress, IEEE1394 operation is kept alive by processing the API callbacks.

When a start event is encountered while a camera is connected, operation is started as follows:
The frame transfer task is disabled to ensure that command processing is finished first. Then the isochronous resources on the input side are allocated (*dcaminIsoAllocateResources*). This is necessary to ensure interoperability with other devices on this IEEE1394 interface and is required

by the IIDC standard. After resource allocation, outgoing streaming is set up by calling *dcamoutIsoTalk*. Then incoming streaming is set up by calling *dcaminIsoListen*. Finally the camera is started by calling *dcaminStartCamera* and the frame transfer task is enabled to run. The latter causes a task switch to the frame transfer task as soon as a complete frame came in.

```
if (bStartEvent == TRUE)
{
    BOOL bError = FALSE;
    /* initialize buffer index */
    u32CurBuf          = 0;
    u32FramesProcessed = 0;
    /* disable frame transfer task until this command has been processed completely */
    TSK_setpri(&TSK_frame_xfer, -1);
    /* allocate isochronous resources */
    Status = dcaminIsoAllocateResources(iPacketSize,
                                        sCurIso.eTransmitSpeed, sCurIso.usIsoChannel);
    if (Status != DCAMIN_SUCCESS)
    {
        fprintf (fid_write, "can't allocate isochronous resources: "
                 "dcaminIsoAllocateResources returned %04Xh\r\n", Status);
        bError = TRUE;
    }
    if (bError == FALSE)
    {
        /* start streaming on output side */
        Status = dcamoutIsoTalk ((Uint32)BUF_Out.startaddr,
                                 BUF_Out.size,
                                 BUF_Out.totalbuffers,
                                 soutVideoData.usIsoChannel,
                                 iPacketSize,/*same as input side*/
                                 soutVideoData.usTransmitSpeed,
                                 &LCK_Out,
                                 &SEM_FrameTx);
        if (Status != DCAMOUT_SUCCESS)
        {
            /* clean up */
            dcaminIsoDeallocateResources(sCurIso.usIsoChannel);
            fprintf (fid_write, "dcamoutIsoTalk() failed %04Xh\r\n", Status);
            bError = TRUE;
        }
    }
    /* start streaming on input side */
    if (bError == FALSE)
    {
        Status = dcaminIsoListen((Uint32)BUF_In.startaddr,
                                 BUF_In.size,
                                 BUF_In.totalbuffers,
                                 OneChannel,
                                 sCurIso.usIsoChannel,
                                 0,
                                 1,
                                 FALSE,
                                 &LCK_In,
                                 &SEM_FrameRx);
        if (Status != DCAMIN_SUCCESS)
        {
            /* clean up */
            Status = dcamoutIsoStop();
            dcaminIsoDeallocateResources(sCurIso.usIsoChannel);
            fprintf (fid_write, "dcaminIsoListen() failed: %04Xh\r\n", Status);
            bError = TRUE;
        }
    }
    /* start camera on input side */
    if (bError == FALSE)
    {
        Status = dcaminStartCamera (hCamera);
        if (Status != DCAMIN_SUCCESS)
        {
            /* clean up */
            Status = dcaminIsoStop(STOP_TIMEOUT);
            Status = dcamoutIsoStop();
            dcaminIsoDeallocateResources(sCurIso.usIsoChannel);
            fprintf (fid_write, "dcaminStartCamera() failed: "%04Xh\r\n", Status);
            bError = TRUE;
        }
        else
        {
            fprintf (fid_write, "Operation started.\r\n");
            /* re-enable frame transfer task */
            TSK_setpri(&TSK_frame_xfer, XFER_TASK_PRIO);
        }
    }
    /* mark event as being processed */
```

```
        bStartEvent = FALSE;
    }
```

When a stop event is encountered, incoming streaming is stopped at the next frame boundary. The camera still sends data, but this data is ignored. Then the camera is stopped (*dcaminStopCamera*) and isochronous resources are de-allocated (*dcaminIsoDeallocateResources*). Before outgoing streaming is also stopped (*dcamoutIsoStop*), the software waits until all buffers have been transmitted to ensure that the last frame is complete. Finally transfer statistics are being displayed.

```
    if (bStopEvent == TRUE)
    {
        fprintf (fid_write, "%d frames processed\r\n", u32FramesProcessed);

        bStopInProgress = TRUE;

        /* stop incoming streaming at the next frame boundary */
        /* this is done first to support cameras that stop outside of frame boundaries */
        Status = dcaminIsoStop(STOP_TIMEOUT);
        if (Status != DCAMIN_SUCCESS)
        {
            fprintf (fid_write, "dcaminIsoStop() failed %04Xh\r\n", Status);
        }

        /* Stop camera */
        Status = dcaminStopCamera (hCamera);
        if (Status != DCAMIN_SUCCESS)
        {
            fprintf (fid_write, "dcaminStopCamera() failed %04Xh\r\n", Status);
        }
        /* de-allocate isochronous resources */
        Status = dcaminIsoDeallocateResources(sCurIso.usIsoChannel);
        if (Status != DCAMIN_SUCCESS)
        {
            fprintf (fid_write, "dcaminDeallocateResources() failed: %04Xh\r\n", Status);
        }
        /* wait until all output buffers are free. This is necessary */
        /* to ensure all buffers get completely transmitted. */
        while(1)
        {
            BOOL bEmpty;
            Uint32 u32Idx;
            for (u32Idx = 0; u32Idx < BUF_In.totalbuffers; u32Idx++)
            {
                dcamoutIsoGetBufferState(u32Idx, &bEmpty);
                if (bEmpty == FALSE)
                    break;
            }
            if (bEmpty == TRUE)
                break;
        }
        /* stop outgoing streaming at the next frame boundary */
        Status = dcamoutIsoStop();
        if (Status != DCAMIN_SUCCESS)
        {
            fprintf (fid_write, "dcamoutIsoStop() failed %04Xh\r\n", Status);
        }
        fprintf (fid_write, "Operation stopped.\r\n");
        bStopInProgress = FALSE;
        bStopEvent = FALSE;
        DisplayTransferStatistics();
    }
```

5.1.4 Frame Processing Task

The frame processing task manages incoming and outgoing transfers on a buffer basis. Buffers are implemented as a DSP/BIOS BUF objects (in VP_example_720MHz_64MB.tcf) with separate buffer objects for input and output and double-buffering for each direction.

```
bios.BUF.create("BUF_In");
bios.BUF.instance("BUF_In").align = 8;
bios.BUF.instance("BUF_In").size = 153600;
bios.BUF.instance("BUF_In").comment = "Input frame buffers";
bios.BUF.instance("BUF_In").bufSeg = prog.get("ERAM_BUFFERS");
bios.BUF.instance("BUF_In").bufCount = 2;
bios.BUF.create("BUF_Out");
bios.BUF.instance("BUF_Out").align = 8;
bios.BUF.instance("BUF_Out").size = 153600;
bios.BUF.instance("BUF_Out").comment = "Output frame buffers";
bios.BUF.instance("BUF_Out").bufSeg = prog.get("ERAM_BUFFERS");
bios.BUF.instance("BUF_Out").bufCount = 2;
```

Two semaphores synchronize the frame processing task with incoming and outgoing streaming. For the input side, *SEM_FrameRx* is posted whenever a complete frame has been received. For the output side, *SEM_FrameRx* is posted whenever a buffer has been completely transmitted. This step is optional, because outgoing streaming must be manually started on a per-buffer basis. The task waits for both semaphores and then checks the state of the current buffer to make sure that it contains new data. The outgoing buffer is locked to make sure it is not transmitted before it is filled. Only then, the image processing function is called. After image processing, the input buffer is released so that it can be filled again. The output buffer is submitted for transmit, which actually starts transmit operation. Finally, transfer statistics are updated.

```
bios.SEM.create("SEM_FrameRx");
bios.SEM.instance("SEM_FrameRx").comment = "Triggers image processing whenever a new frame was
received";
bios.SEM.create("SEM_FrameTx");
bios.SEM.instance("SEM_FrameTx").comment = "Triggered whenever a frame has been sent";
```

```
int FXN_frame_xfer(void)
{
    while(1)
    {
        BOOL bFilled;
        BOOL bEmpty;
        DCAM_STATUS Status;    /* return status of DCAM API calls */

        /* wait for incoming frame */
        SEM_pend(&SEM_FrameRx, SYS_FOREVER);

        /* wait for a free output buffer (should always be the case) */
        SEM_pend(&SEM_FrameTx, SYS_FOREVER);

        /* To be absolutely sure, check the buffer states */
        Status = dcaminIsoGetBufferState(u32CurBuf, &bFilled);
        if (Status != DCAMIN_SUCCESS || bFilled == FALSE)
        {
            fprintf (fid_write, "Input buffer error\r\n");
        }
        Status = dcamoutIsoGetBufferState(u32CurBuf, &bEmpty);
        if (Status != DCAMOUT_SUCCESS || bEmpty == FALSE)
        {
            fprintf (fid_write, "Output buffer error\r\n");
        }
        /* lock output buffer for image processing */
        Status = dcamoutIsoLockBuffer(u32CurBuf);
        if (Status != DCAMOUT_SUCCESS)
        {
            fprintf (fid_write, "Can't lock output buffer\r\n");
        }
        /* call image processing */
        ProcessImage(&BUF_In, &BUF_Out, u32CurBuf);
        /* release input buffer to be filled again */
        Status = dcaminIsoReleaseBuffer(u32CurBuf);
        if (Status != DCAMOUT_SUCCESS && bStopInProgress == FALSE)
        {
            fprintf (fid_write, "Can't release input buffer\r\n");
        }
        /* submit output buffer for transmit */
        Status = dcamoutIsoSubmitBuffer(u32CurBuf);
        if (Status != DCAMOUT_SUCCESS)
        {
            fprintf (fid_write, "Can't submit output buffer\r\n");
        }

        /* select next buffers */
        u32CurBuf++;
        u32FramesProcessed++;
        if (u32CurBuf >= BUF_In.totalbuffers)
            u32CurBuf = 0;
    }
}
```

Incoming streaming is configured to run autonomously as a set of linked EDMA transfers. Only a transfer complete handler is called when a frame is complete. Overruns are detected by this transfer handler and are displayed by the main control task when operation is stopped.

```
Camera stop command received.
1609 frames processed
Operation stopped.
transfer statistics:
```

```
Buffers received/transmitted:       1610       1610
Buffers released/submitted:         1609       1610
Double-buffer runs in/out:           805        805
Buffer errors in/out:                  0          0
Last error in buf #                    0          0
Restart errors in/out:                 0          0
Last error in buf #                    0          0
Overflow errors in/out:                0          0
Last overflow in buf #                 0          0
Underflow errors in/out:               0          0
Last underflow in buf #                0          0
DM port errors in/out:                 0          0
Last DM port error in buf #            0          0
1394 ApiErrors in/out:                 0          0
Last 1394 Api error code               0          0
Synchronization errors:                0
Last sync error in buf #               0
```

Checking for errors during operation can be done by calling *dcaminIsoGetStatistics* and *dcamoutIsoGetStatistics* and checking the error counters.

Using common buffers for image processing is also possible for some algorithms and may give better performance due to the fact that the cache does not need to hold input and output locations separately. In this case, please refer to section 5.1.10 for modification instructions.

At startup, the frame processing task is disabled by assigning priority -1 in the DSP/BIOS configuration script.

```
bios.TSK.create("TSK_frame_xfer");
bios.TSK.instance("TSK_frame_xfer").order = 2;
bios.TSK.instance("TSK_frame_xfer").comment = "frame buffer management and image processing";
bios.TSK.instance("TSK_frame_xfer").priority = -1;
bios.TSK.instance("TSK_frame_xfer").fxn = prog.extern("FXN_frame_xfer");
```

When the first start command is executed by the main control task, the frame transfer task is started, executes *SEM_pend* and gets suspended again until the first frame has been received. After reception of the first frame, the task is activated and calls *SEM_pend* for the output buffer. This call immediately returns, because all buffers are initially available and *SEM_post* has been called for each buffer during execution of *dcamoutIsoTalk* in the main control task, when starting operation.

5.1.5 Image Processing

The image processing function (*ProcessImage*) gets both buffer objects and the current buffer index as parameters. The current algorithm compares the Y part of each pixel against the respective pixel of the previous image. If there is a change in intensity that is larger than 12.5%, the pixel is left unmodified. If the pixel's intensity didn't change, the Y part is reduced by 50%. During execution of the image processing function, the red LED is switched on as an optical activity indicator.

```
void ProcessImage(BUF_Handle hSrc, BUF_Handle hDst, Uint32 u32BufIdx)
{
#if 0 /* direct route-through for testing */
    memcpy((void *)((Uint32)hDst->startaddr + u32BufIdx * hDst->size),
           (void *)((Uint32)hSrc->startaddr + u32BufIdx * hSrc->size),
           hSrc->size);
#else
    /* check if there is a significant intensity change between previous and */
    /* current frame. If there is no significant change, dim the intensity */
    /* by 50%. This should have the effect of highlighting changes */
    Uint32 u32PrevBufIdx;
    Uint16 *pu16Cur;  /* pointer to current input frame's data */
    Uint16 *pu16Prev; /* pointer to last frame */
    Uint16 *pu16Dst;  /* pointer to output frame's data */
    Uint16 u16Pix1;   /* UY pixel */
    Uint16 u16Pix2;   /* VY pixel */
    Uint8  u8Delta1, u8Delta2;
    int    i;

    /* switch on red LED during processing as optical load indicator */
    C6412CPT_RED_LED_ON;

    /* calculate buffer index for previous frame */
    if (u32BufIdx > 0)
        u32PrevBufIdx = u32BufIdx - 1;
```

```
    else
        u32PrevBufIdx = hSrc->totalbuffers - 1;

    /* set up pointers to the image buffers */
    pu16Cur =  (Uint16 *)((Uint32)hSrc->startaddr + u32BufIdx     * hSrc->size);
    pu16Prev = (Uint16 *)((Uint32)hSrc->startaddr + u32PrevBufIdx * hSrc->size);
    pu16Dst =  (Uint16 *)((Uint32)hDst->startaddr + u32BufIdx     * hDst->size);

    /* processing loop in units of 2 pixels (UY pixel followed by VY pixel, */
    /* 4 byte total) */
    for (i = 0; i < hSrc->size / 4; i++)
    {
        /* Get 2 pixel of input data. The byte locations in memory are as */
        /* defined in the IIDC spec (U-Y-V-Y-U-Y-...). However, due to the */
        /* 16-bit read and the little-endian architecture, Y gets into the */
        /* MSB and 'U/V into the LSB */
        u16Pix1  = *pu16Cur++;
        u16Pix2  = *pu16Cur++;
        /* calculate intensity difference between current and previous frame */
        /* for each pixel */
        u8Delta1 = (u16Pix1 >> 8) - (*pu16Prev++ >> 8);
        u8Delta2 = (u16Pix2 >> 8) - (*pu16Prev++ >> 8);
        /* dim intensity if difference to previous frame doesn't exceed */
        /* threshold */
        if (u8Delta1 < POS_LIMIT || u8Delta1 > NEG_LIMIT)
            u16Pix1 = ((u16Pix1 / 2) & 0xFF00) | (u16Pix1 & 0x00FF);
        if (u8Delta2 < POS_LIMIT || u8Delta2 > NEG_LIMIT)
            u16Pix2 = ((u16Pix2 / 2) & 0xFF00) | (u16Pix2 & 0x00FF);
        /* write to output buffer */
        *pu16Dst++ = u16Pix1;
        *pu16Dst++ = u16Pix2;
    }

    C6412CPT_RED_LED_OFF;
#endif
}
```

For testing, the unmodified image can be enabled by changing the line

```
#if 0 /* direct route-through for testing */
```

to

```
#if 1 /* direct route-through for testing */
```

5.1.6 Output-side Setup

Setting up the output side is implemented in file *camera_setup.c*. Two objects that are defined in *main.c* must be referenced: the file pointer for debug message output and the video settings structure:

```
// for debug output
extern FILE *fid_write;

// output side settings are in main.c
extern DCAMOUTs_VIDEO_DATA soutVideoData;
```

The function *SetupDigitalCamera* performs all necessary setup. Parts of the setup have been moved to separate functions to keep this function straightforward.
First part is to set up the supported video settings (format, mode and frame rate) with function *SetupVideoInquiryRegs*. The example implements only one single combination of video settings to avoid switching between different settings. The supported settings are 320x200, YUV 4:2:2, 30fps. This setting is compatible with the built-in drivers of Windows XP.
Next part is setting up the supported basic functions with function *SetupBasicFuncInquiryReg*. The only basic function implemented by the example is the video mode error status register. This register allows the camera controller to check for errors after setting up the video settings, but before starting operation. The video mode error status register is handled internally by the DCAM Camera API. Please note that the IIDC standard implicitly requires to have memory channel 0 implemented, which is used for restoring to factory settings. The example, however doesn't implement this function, because there are no settings to change.
Third step is setup of the initial camera control & status registers with function *InitCameraStatCtrlRegs*. Setting these values allows a camera controller to directly start operation without prior setup. However, in most cases the camera controller will do a complete setup and overwrite these defaults. Default settings are

- video format, mode and frame rate as previously set up
- isochronous channel 0
- isochronous transmit speed of 400Mbps.

Up to now, only the minimum required setup was implemented. The remaining steps implement two camera features: brightness and sharpness. These features do not have any influence on the image, but rather demonstrate feature implementation and indicate feature access by callback functions. First part of feature setup is setting up the features and their capabilities (function *SetupFeatureInquiryRegs*). Then, the initial feature settings are set up by function *InitFeatureStatCtrlRegs*. Next step is installing a common callback function for feature write accesses. This callback will generate messages when a feature is changed by the camera controller. Final step is to install all feature-related registers.

5.1.7 Output-side Callbacks

Operation of the output side requires the following callback functions to be installed:

- *InitializeCallback* should initialize the output side to power-up defaults. Currently, only a message is printed.
- *VideoSettingsCallback* should set up operation according to the parameters written by the camera controller. Depending on the application this may involve:
  - Modifying the image processing algorithm to generate a different output format.
  - Modifying the settings of the input side, thus forwarding the video setting to the camera used as image source.
- *IsoEnableCallback* This callback controls start and stop of operation by setting the appropriate event flags *bStartEvent* and *bStopEvent*. These flags are polled in the main control task, where the appropriate action is performed and the flags are reset. Please note that this mechanism must be implemented, because API calls from within a callback function are not allowed and callback execution time should be minimized.
- *FeatureStatCtrlCallback* should take the appropriate action depending on the accessed feature. For example, when the camera controller increases brightness, the callback could set an event flag that causes the new brightness value to be forwarded to the connected camera. Or the callback could update a variable that controls the output intensity of the image processing algorithm.

5.1.8 Resource Usage

VP_example uses the typical resources occupied by DSP/BIOS, such as a timer plus the following:

- Both IEEE1394 interfaces
- RS-232 for message output
- External interrupts 4 and 5 for operation of the IEEE1394 APIs
- GP0[11] and GP0[12] for triggering EDMA transfers (available for measurement on connector pins B23 and B24)
- EDMA channel 51 for the input side
- EDMA channel 52 for the output side
- 4 EMDA reload channels (one for each frame buffer)
- End of EDMA interrupt (using the CSL's interrupt dispatcher that allows additional interrupts on other channels)
- DSP/BIOS objects as listed in section 4.4.7 and following.

5.1.9 Performance

When running on a C6412Compact with 720MHz CPU clock, the image processing function of *VP_example* takes roughly 11ms in debug configuration and 3ms in Release configuration. Further speed-up may be achieved by using in-place image processing, where the same buffer is used for input and output. However, many image processing algorithms, including the one of *VP_example*, require separate input and output buffers.

5.1.10 Modifying for In-place Processing

Modifying *VP_example* for in-place (common frame buffers) operation can speed up image processing, but requires a suitable image processing algorithm.

- Remove one of the frame buffer objects and rename the other to some common name. This must be done in
  - the DSP/BIOS configuration (*.tcf*) file
  - the declaration, definition and call of the image processing function
  - the image processing function itself
  - API functions *dcaminIsoListen* and *dcamoutIsoTalk*
  - stop event handling (buffer loop that waits for all buffers to get sent)
- Add a separate buffer index for the output side:

```
static Uint32            u32CurBuf;
static Uint32            u32CurBufRelease;
…
                /* initialize working variables */
                u32CurBuf          = 0;
                u32CurBufRelease   = 0;
                u32FramesProcessed = 0;
```

- Modify the frame processing task as follows:
  - Remove the call to *SEM_pend* for the output buffer
  - Remove the call to *dcaminIsoReleaseBuffer*
- Add a separate task (in the DSP/BIOS configuration file and in *main.c*):

```
int FXN_frame_release(void)
{
    while(1)
    {
        DCAM_STATUS Status;    /* return status of DCAM API calls */

        /* wait for incoming frame */
        SEM_pend(&SEM_FrameTx, SYS_FOREVER);
        /* release buffer so that it can be filled again */
        if (bStopInProgress == FALSE && u32FramesProcessed > 0)
            Status = dcaminIsoReleaseBuffer(u32CurBufRelease);
        if (Status != DCAMOUT_SUCCESS)
            fprintf (fid_write, "Can't release input buffer\r\n");
        u32CurBufRelease++;
        if (u32CurBufRelease >= BUF_Frame.totalbuffers)
            u32CurBufRelease = 0;
    }
}
```

5.1.11 Control and Data Flow

When the *VP_example* is used as described in chapter 3, the following sequence of events and actions occurs:

- The camera controller looks for available IIDC devices and finds the C6412Compact.
- The camera controller sends some commands to the C6412Compact to adjust some settings. These commands are reported by the associated callback functions.
- The C6412Compact looks for available IIDC devices on its IEEE1394 interface 1 and finds the camera.
- The C6412Compact sends some commands to the camera to adjust some settings.
- When the user starts camera operation (e.g. clicking on the "Generic 1394 Desktop Camera" icon), the camera controller allocates isochronous resources and sends some commands to the C6412Compact.
- The software on the C6412Compact recognizes the commands, allocates isochronous resources for the input side and starts isochronous operation on both, input and output side. Then, it sends a start command to the camera.
- The camera starts transmitting an isochronous data stream. The first packet has a sync bit set in its packet header. This sync bit causes the interface logic in the C6412Compact's FPGA to start reception.
- The incoming isochronous data is buffered in a FIFO in the FPGA. Whenever enough data is collected, the FPGA triggers an EDMA transfer via GP0[11]. This transfers the image data to the current frame buffer.

- When a complete buffer has been received, an event is generated by the DCAM frame capture library that triggers the frame processing task.
- The frame processing task calls the image processing function.
- After image processing has finished, the frame processing task submits the output frame buffer for transmit and also starts transmission. The DCAM Camera library sets a flag in the FPGA hardware that marks the next data word as the first of a frame (for sync bit insertion).
- Whenever the transmit FIFO in the FPGA has enough space, the FPGA triggers an EDMA transfer via GP0[12]. This transfers the processed image data from the output frame buffer to the transmit FIFO in the FPGA. The first data word of each frame has a flag set, that causes sync bit insertion in the first packet's header, so that the receiver can synchronize to frame start.
- The image data is transferred from the transmit FIFO to IEEE1394 interface 2 in 1394 packets.
- The camera controller receives the image data, synchronizes to frame start and displays the image data.

6 System-specific Data Types

Some data types of the DCAM frame capture and the DCAM camera APIs are specific to the features of the respective hardware. These types are described in this chapter.

6.1 Incoming Streaming Statistics

This data type is the same as *S_STR_RX_STATISTICS*, described in [5]. It is defined as

```
/* transfer and error statistics */
/* for each error the buffer number where the last error occured is stored */
/* except for IEEE1394 API errors, where the last error code is stored */
typedef struct
{
    /* transfer statistics */
    INT32U     u32Received;
    INT32U     u32Released;
    INT32U     u32Runs;
    /* error statistics */
    INT32U     u32BufErrors;
    INT32U     u32BufLastBuf;
    INT32U     u32RestartErrors;
    INT32U     u32RestartLastBuf;
    INT32U     u32OvErrors;
    INT32U     u32OvLastBuf;
    INT32U     u32UnErrors;
    INT32U     u32UnLastBuf;
    INT32U     u32DmErrors;
    INT32U     u32DmLastBuf;
    INT32U     u32ApiErrors;
    INT32U     u32ApiLastError;
    INT32U     u32SyncErrors;
    INT32U     u32SyncLastBuf;
} S_DCAMIN_STATISTICS;
```

6.2 Outgoing Streaming Statistics

This data type is the same as *S_STR_TX_STATISTICS*, described in [5]. It is defined as

```
/* transfer and error statistics */
/* for each error the buffer number where the last error occured is stored */
/* except for IEEE1394 API errors, where the last error code is stored */
typedef struct
{
    /* transfer statistics */
    Uint32     u32Submitted;
    Uint32     u32Transmitted;
    Uint32     u32Runs;
    /* error statistics */
    Uint32     u32BufErrors;
    Uint32     u32BufLastBuf;
    Uint32     u32RestartErrors;
    Uint32     u32RestartLastBuf;
    Uint32     u32OvErrors;
    Uint32     u32OvLastBuf;
    Uint32     u32UnErrors;
    Uint32     u32UnLastBuf;
    Uint32     u32DmErrors;
    Uint32     u32DmLastBuf;
    Uint32     u32ApiErrors;
    Uint32     u32ApiLastError;
} S_DCAMOUT_STATISTICS;
```

7 System-specific Functions

Some functions of the DCAM frame capture and the DCAM camera APIs are specific to the features of the respective hardware. These functions are described in this chapter. They include

- API callback queue
- incoming isochronous streaming
- outgoing isochronous streaming

7.1 API Callback Queue Functions

The DCAM Camera API and the DCAM Frame Capture API are based on an IEEE1394 embedded API each. The IEEE1394 embedded APIs use (for historical reasons) a common callback queue where requests from the 1394 buses are queued in form of callback functions. Requests can result from either user calls to API functions or from events, such as incoming transactions or bus resets. The functions described in this section handle the call queue and are defined in *callq.h*. The function *InitCallQ* must be called at startup to initialize the call queue. After start of 1394 processing on the input or output bus, the application software should call the function *YieldToCallQ* at least every 20 ms.

7.1.1 InitCallQ

InitCallQ initializes the callback queue that is used by the IEEE1394 APIs. This function must be called once during system initialization before the DCAM APIs are initialized.

defined in
callq.h

synopsis
```
void InitCallQ(void);
```

parameters
none

return value
none

7.1.2 YieldToCallQ

Processing of the callback queue. *YieldToCallQ* processes one callback from the queue (if present) by removing it from the queue and calling it. The DCAM APIs and the underlying IEEE1394 APIs are not reentrant; therefore, this function must not be called from within a callback function or an interrupt handler. If multithreading is used, *YieldToCallQ* as well as API functions must be called from within a single thread only unless otherwise noted. During operation, *YieldToCallQ* must be called at least every 20ms to avoid timeouts.

defined in
callq.h

synopsis
```
void InitCallQ(void);
```

parameters
none

return value
none

7.2 Incoming Isochronous Streaming Functions

The functions for incoming isochronous streaming handle the incoming data stream from one or more cameras. The incoming data is first buffered by the streaming cores[7] in the FPGA and then transferred to the frame buffers using EDMA transfers. For each transferred frame buffer an event is generated that can be used to synchronize an image processing task. The functions can be grouped into the following categories:

- Initialization: *dcaminIsoInit*
- Resource management: *dcaminIsoAllocateResources*, *dcaminIsoDeallocateResources*
- Transfer control: *dcaminIsoListen*, *dcaminIsoStop*
- Buffer management: *dcaminIsoGetBufferState*, *dcaminIsoReleaseBuffer*
- Statistics: *dcaminIsoGetStatistics*, *dcaminIsoGetLastError*

Internally, the incoming isochronous functions use the software streaming libraries (see [5] for a description) with the following settings:

- automatic restart (continuous receive operation without software intervention required)
- endian swapping (fits the little endian architecture of the C6412Compact)
- automatic block size determination (use minimum number of EDMA transfers with maximum size)
- EDMA transfer trigger on GP0[11], which is available on pin B23 for measurements
- PDT transfers disabled (not yet qualified)
- cache management: after a frame buffer has been received, an L2 block invalidate is performed on the frame buffer before the buffer-received event is posted.

All other settings are user-definable or have fixed settings, required by the IIDC standard [15]. The functions are declared in a separate header file, *dcaminIso.h*, which is included by *dcaminApi.h*. This allows hardware-specific implementation. Applications should not include *dcaminIso.h* directly but include *dcaminApi.h* instead.
When using multithreading, care must be taken where the functions can be called from. Functions that can cause IEEE1394 transactions must be called in the main application thread where all other DCAM API functions are called, whereas all other functions can be called from a different thread. The table below shows where each function can be called from:

Function	To be called in
dcaminIsoInit	system initialization function (*main.c*)
dcaminIsoAllocateResources	main application thread only
dcaminIsoDeallocateResources	main application thread only
dcaminIsoListen	main application thread only
dcaminIsoStop	main application thread only
dcaminIsoGetBufferState	no restriction, typically called in image processing thread
dcaminIsoReleaseBuffer	no restriction, typically called in image processing thread
dcaminIsoGetStatistics	no restriction
dcaminIsoGetLastError	no restriction

Table 1: Function callability table for incoming isochronous streaming

7.2.1 Incoming Isochronous Streaming Data Types

```
typedef struct
{
    /* transfer statistics */
    INT32U     u32Received;
    INT32U     u32Released;
    INT32U     u32Runs;
    /* error statistics */
    INT32U     u32BufErrors;
```

```
    INT32U      u32BufLastBuf;
    INT32U      u32RestartErrors;
    INT32U      u32RestartLastBuf;
    INT32U      u32OvErrors;
    INT32U      u32OvLastBuf;
    INT32U      u32UnErrors;
    INT32U      u32UnLastBuf;
    INT32U      u32DmErrors;
    INT32U      u32DmLastBuf;
    INT32U      u32ApiErrors;
    INT32U      u32ApiLastError;
    INT32U      u32SyncErrors;
    INT32U      u32SyncLastBuf;
} S_DCAMIN_STATISTICS;
```

Field	Description
u32Received	Number of completely received frames.
u32Released	Number of frame buffers released (=processed by software)
u32Runs	Number of completed buffer sequences (u32Received divided by number of buffers)
u32BufErrors	Number of buffers that were not marked as empty after being filled[3].
u32BufLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32RestartErrors	(currently unused)
u32RestartLastBuf	(currently unused)
u32OvErrors	Number of receive FIFO overflow conditions in the streaming core.
u32OvLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32UnErrors	Number of receive FIFO underflow conditions in the streaming core.
u32UnLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32DmErrors	Number of data mover port errors signaled from the IEEE1394 chipset.
u32DmLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32ApiErrors	Number of errors returned from the IEEE1394 API subsystem.
u32ApiLastError	Last error code of the IEEE1394 API subsystem.
u32SyncErrors	Streaming core FIFO not empty at end of frame.
u32SyncLastBuf	Number of the buffer (starts at 0) where the above error occurred.

7.2.2 dcaminIsoInit

dcamIsoInit clears internal variables and initializes the streaming subsystem. This function must be called once during system startup.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoInit(void);
```

parameters
none

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_INIT	See chapter 7.4

7.2.3 dcaminIsoAllocateResources

The IIDC specification[15] requires that a camera controller allocates the required isochronous resources occupied by the data stream from the camera. These are a specific amount of bus

[3] There is currently no way to check for an empty buffer at the beginning of a frame, there fore the check must be done at the end.

bandwidth and a channel number. Bus bandwidth depends on the packet size and the transmit speed. A packet transferred at S100 uses 4x the bandwidth as the same packet transmitted at S400. *dcaminIsoAllocateResources* must be called before isochronous receive operation is started (before calling *dcamIsoListen*). If more than one camera is connected, resource allocation must be done separately for each camera. The DCAM Frame Capture API internally keeps a list of allocated bandwidth/channel resource pairs for later de-allocation.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoAllocateResources(Uint32 u32BytesPerPkt, Uint32 u32Speed, Uint8 u8Channel);
```

parameters

u32BytesPerPkt	Requested bandwidth in terms of isochronous packet size. Valid range is 4 to 4096. The value is rounded up to the next multiple of 4 if necessary. Packet size depends on format / mode / frame rate and can be queried by *dcaminQueryPacketSize* for formats 0, 1 and 2. For format 7, u32BytesPerPkt must be the same as used for *dcaminSetCameraF7PacketSize*.
u32Speed	Isochronous transfer speed as used for *dcaminSetIsoData*. Valid values are S100, S200 and S400.
u8Channel	Isochronous channel number as used for *dcaminSetIsoData*. Valid range is 0 to 63, where channel 31 is reserved by [17] and should not be used.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_RESOURCE	See chapter 7.4

7.2.4 dcaminIsoListen

dcaminIsoListen starts incoming streaming operation. After calling this function, incoming isochronous data is stored in the specified frame buffers, events are generated for completely received buffers and buffer state can be queried with *dcaminIsoGetBufferState*. Please note that this function does not start connected cameras. Typically, *dcaminIsoListen* is called before *dcaminStartCamera* is called.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoListen(INT32U u32BufferBase, INT32U u32BufferSize, INT32U u32BufferCount,
                            DCAMINe_ISO_LISTENMODE eMode, INT16S iFirstChannelNo,
                            INT16S iSecondChannelNo, INT16S iSyncBits, BOOL bHeader,
                            LCK_Handle hLck, SEM_Handle hSem);
```

parameters

u32BufferBase	Start address of the input frame buffers. The input frame buffers must be located completely within SDRAM or in internal DSP memory. Typically, buffers are located in SDRAM. Start address must be aligned on an 8 byte boundary.
u32BufferSize	Size of one frame buffer in bytes. Frame size can be determined by *dcaminQueryFrameSize* for formats 0, 1 and 2. For format 7, frame size depends on the parameters passed to *dcaminSetCameraFormat7Image*.
u32BufferCount	Number of input frame buffers. The number of buffers is limited by SDRAM size and the number of available EDMA channels (around 70 per direction).
eMode	Receive mode. Possible settings are *OneChannel*, *TwoChannel*, *AllChannel*, according to the number of cameras used. For 3 and more cameras, *AllChannel* must be used and no other isochronous data streams must be active.
iFirstChannelNo	Isochronous channel number for single-camera operation or channel number for first camera for dual-camera operation. Ignored for multiple (3 or more) camera operation. Must be set to the channel number that was used as parameter for *dcaminSetIsoData*.
iSecondChannelNo	Isochronous channel number for dual-camera operation. Ignored for single-camera and multiple (3 or more) camera operation. Must be set to the channel number that was used as parameter for *dcaminSetIsoData*.
iSyncBits	Synchronization pattern for frame start. Must be set to 1. This parameter may be removed in future versions.
bHeader	Controls whether packet header information (*data_length*, *tg*, *channel*, *tCode* and *sy* fields) are included in the data stream. For single camera operation, *bHeader* should be set to *FALSE*. For operation with more than one camera, *bHeader* should be set to *TRUE*, so that the incoming data packets can be associated with a specific camera by the *channel* field of the packet header.
hLck	DSP/BIOS resource lock handle for IEEE1394 interface 1. Typically, *hLck* refers to a LCK object, which is defined in the DSP/BIOS configuration script (*.tcf* file) of the video processing application. *hLck* is used internally by the API to lock against multiple accesses to the hardware.
hSem	DSP/BIOS semaphore handle for generating incoming frame buffer events. *hSem* will be posted whenever a frame has been received and is ready for processing. Typically, *hSem* refers to a SEM object, which is defined in the DSP/BIOS configuration script (*.tcf* file) of the video processing application. If frame buffer events are not required, *hSem* can be set to NULL.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_INVALID_PARAMETER	*eMode*, *iFirstChannelNo* or *iSecondChannelNo* are out of range.
DCAMIN_ERROR_ISO_START	See chapter 7.4

7.2.5 dcaminIsoGetBufferState

While isochronous receive is active, *dcaminGetBufferState* can be used to check the state of a specific frame buffer. Typically, this is only used for diagnostic purposes and image processing is triggered using a semaphore.

defined in
dcaminIso.h

synopsis
```
DCAM_STATUS dcaminIsoGetBufferState(INT32U u32Buffer, BOOL *bFilled);
```

parameters

u32buffer	Number of the buffer to be queried. Valid range is 0 to *u32BufferCount* - 1, where *u32BufferCount* was used as parameter for *dcaminIsoListen*.
bFilled	State of the specified buffer, set by *dcaminIsoGetBufferState* after return. *TRUE* indicates that the buffer is filled and ready for image processing. After image processing, buffer state should be updated by calling *dcaminIsoReleaseBuffer*. *FALSE* indicates that the buffer has not yet been completely filled.

return value

DCAMIN_SUCCESS	Successful completion, *bFilled* contains valid information.
DCAMIN_ERROR_ISO_BUFSTAT	Status query returned an error and *bFilled* is set to *FALSE*. See chapter 7.4 for a list of possible reasons.

7.2.6 dcaminIsoReleaseBuffer

After an input buffer has been processed (and transmitted if in-place image processing is used, see chapter 5.1.10), it must be released so that it can be filled again. In the current implementation, the buffer will be filled anyway, but an error is generated (after filling the buffer) if the buffer is already marked as filled.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoReleaseBuffer(INT32U u32Buffer);
```

parameters

u32buffer	Number of the buffer to be released. Valid range is 0 to *u32BufferCount* - 1, where *u32BufferCount* was used as parameter for *dcaminIsoListen*.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_BUFFER	See chapter 7.4.

7.2.7 dcaminIsoStop

dcaminIsoStop stops incoming streaming operation at the next frame buffer boundary. This ensures that receive operation currently in progress is finished. If the camera continues to transmit image data, it will be ignored. If no image data is being transmitted when this function is called, the function will time out and return with error. Transfer statistics are stored for later retrieval by *dcaminIsoGetStatistics*.

defined in
dcaminApi.h

synopsis

```
DCAM_STATUS dcaminIsoStop (INT32U u32Timeout);
```

parameters

u32Timeout	Timeout for stopping reception in milliseconds. Typically, this value should be set to the nominal frame time (e.g. 33ms for 30fps) plus some margin, e.g. 10%.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_STOP	See chapter 7.4.

7.2.8 dcaminIsoDeallocateResources

dcamIsoDeallocateResources de-allocates the isochronous resources used by a previous operation. This allows other IEEE1394 devices to use these resources. This is especially important if other devices (than camera and C6412Compact) are connected to the input side (IEEE 1394 interface 1). De-allocation must only be called after isochronous transmission was stopped by *dcaminStopCamera*. Optionally, application software can make sure that no more data is being transmitted, e.g. by waiting for one frame's time. If multiple cameras are being used, de-allocation must be done separately for each camera.

defined in
dcaminApi.h

synopsis

```
DCAM_STATUS dcaminIsoDeallocateResources(Uint8 u8Channel);
```

parameters

u8Channel	Isochronous channel number that was used when allocating the resources (as parameter to *dcaminIsoAllocateResources*). The channel number is used to identify the bandwidth/channel resource pair in the internal resource list of the API.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_RESOURCE	See chapter 7.4.

7.2.9 dcaminIsoGetStatistics

During operation, transfer statistics are collected. These can be queried during or after operation to determine the number of processed frames or to check for errors.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoGetStatistics(S_DCAMIN_STATISTICS *psStat);
```

parameters

psStat	pointer to statistics structure which is initialized by *dcaminIsoGetStatistics*. For a description of this structure see chapter 6.1.

return value

DCAMIN_SUCCESS	Successful completion.
DCAMIN_ERROR_ISO_STATISTICS	See chapter 7.4.

7.2.10 dcaminIsoGetLastError

Internally, the isochronous functions call functions of the IEEE1394 API of the software streaming library. Error codes returned by these functions are stored and can be retrieved by *dcaminGetLastError* for more detailed information about an error. The return value is the status code of the last-called internal API function.

defined in
dcaminApi.h

synopsis
```
DCAM_STATUS dcaminIsoGetLastError(void);
```

parameters
none

return value

DCAMIN_SUCCESS	Successful completion.
0000 … 0FFFh	Error code from IEEE1394 API.
F000 … FFFFh	Error code from software streaming library, see [5].

7.3 Outgoing Isochronous Streaming Functions

The functions for outgoing isochronous streaming handle transmission of the processed image data stream on IEEE1394 interface 2. The data is transferred to the transmit FIFO of the streaming core[7] in the FPGA using EDMA transfers. Application software must trigger transmit operation manually for each frame buffer. This avoids problems with timing differences between the two IEEE1394 interfaces. Whenever a frame buffer has been completely transmitted (to the streaming core), an event is generated that can be used to synchronize an image processing task. The functions can be grouped into the following categories:

- Initialization: *dcamoutIsoInit*
- Transfer control: *dcamoutIsoTalk*, *dcamoutIsoStop*
- Buffer management: *dcamoutIsoGetBufferState*, *dcamoutIsoLockBuffer*, *dcamoutIsoSubmitBuffer*

- Statistics: *dcaminIsoGetStatistics*

Internally, the outgoing isochronous functions use the streaming software package[5] with the following settings:

- single-buffer restart (transmission of each frame buffer must be started manually)
- no allocation of isochronous resources (must be done by the camera controller)
- endian swapping (fits the little endian architecture of the C6412Compact)
- automatic block size determination (use minimum number of EDMA transfers with maximum size)
- EDMA transfer trigger on GP0[12], which is available on pin B24 for measurements
- PDT transfers disabled (not yet qualified)
- Cache management: before a frame buffer is transmitted, a L2 block write back is performed on the frame buffer.

All other settings are user-definable or have fixed settings, required by the IIDC standard[15]. The functions are declared in a separate header file, *dcamoutIso.h*, which is included by *dcamoutApi.h*. This allows hardware-specific implementation. Applications should not include *dcamoutIso.h* directly but include *dcamoutApi.h* instead.

When using multithreading, care must be taken where the functions can be called from. Functions that can cause IEEE1394 transactions must be called in the main application thread where all other DCAM API functions are called, whereas all other functions can be called from a different thread. It is important to note that *dcamoutIsoTalk*/*Stop* functions (as all DCAM API functions) must not be called from within API callbacks. The table below shows where each function can be called from:

Function	To be called in
dcamoutIsoInit	system initialization function (*main.c*)
dcamoutIsoTalk	main application thread only
dcamoutIsoStop	main application thread only
dcamoutIsoGetBufferState	no restriction, typically called in image processing thread
dcamoutIsoLockBuffer	no restriction, typically called in image processing thread
dcamoutIsoSubmitBuffer	no restriction, typically called in image processing thread
dcamoutIsoGetStatistics	no restriction

Table 2: Function callability table for outgoing isochronous streaming

7.3.1 Outgoing Isochronous Streaming Data Types

The structure *S_DCAMOUT_STATISTICS* is used by function *dcamoutIsoGetStatistics* to return transfer statistics.

Definition:

```
struct {
    Uint32      u32Submitted;
    Uint32      u32Transmitted;
    Uint32      u32Runs;
    /* error statistics */
    Uint32      u32BufErrors;
    Uint32      u32BufLastBuf;
    Uint32      u32RestartErrors;
    Uint32      u32RestartLastBuf;
    Uint32      u32OvErrors;
    Uint32      u32OvLastBuf;
    Uint32      u32UnErrors;
    Uint32      u32UnLastBuf;
    Uint32      u32DmErrors;
    Uint32      u32DmLastBuf;
    Uint32      u32ApiErrors;
    Uint32      u32ApiLastError;
} S_DCAMOUT_STATISTICS;
```

Field	Description
u32Submitted	Number of frame buffers submitted for transmit

u32Transmitted	Number of completely transferred frame buffers
u32Runs	Number of completed buffer sequences (u32Transmitted divided by number of buffers)
u32BufErrors	Number of buffers that were not marked as filled after being sent.
u32BufLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32RestartErrors	Streaming transmit core was unexpectedly active when trying to restart.
u32RestartLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32OvErrors	Number of transmit FIFO overflow conditions in the streaming core.
u32OvLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32UnErrors	Number of Transmit FIFO underflow conditions in the streaming core.
u32UnLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32DmErrors	Number of data mover port errors signaled from the IEEE1394 chipset.
u32DmLastBuf	Number of the buffer (starts at 0) where the above error occurred.
u32ApiErrors	Number of errors returned from the IEEE1394 API subsystem.
u32ApiLastError	Last error code of the IEEE1394 API subsystem.

7.3.2 dcamoutIsoInit

dcamoutIsoInit clears statistics and call the initialization function of the streaming software package. This function must be called once at system startup.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoInit(void);
```

parameters
none

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_ISO_INIT	See chapter 7.4.

7.3.3 dcamoutIsoTalk

dcamoutIsoTalk sets up outgoing streaming operation. This function must be called before processed image data can be transmitted by calling *dcamoutIsoSubmitBuffer*. Typically, *dcamoutIsoTalk* is called as a result from an incoming start command from a camera controller. No data is actually transmitted as result of this call. Instead, each frame buffer must be transmitted with *dcamoutIsoSubmitBuffer*.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoTalk (Uint32 u32BufferBase, Uint32 u32BufferSize, Uint32 u32BufferCount,
                            INT16U usIsoChannel, INT16U usPacketSize, INT16U usTransmitSpeed,
                            LCK_Handle hLck, SEM_Handle hSem);
```

parameters

u32BufferBase	Start address of the output frame buffers. The output frame buffers must be entirely located within SDRAM or in internal DSP memory. Must be aligned on an 8 byte boundary. Typically, buffers are located in SDRAM.
u32BufferSize	Size of one frame buffer in bytes. The required buffer size for format 0, 1, 2 can be queried by *dcamoutGetFrameSize*. For format 7, frame buffer size

	is configured by the camera controller and is available in IIDC registers *TOTAL_BYTES_LO/HI_INQ*. If buffer size is not a multiple of 8, it must be rounded up to the next multiple of 8 bytes.
u32BufferCount	Number of input frame buffers. The number of buffers is limited by SDRAM size and the number of available EDMA channels (around 70 per direction).
usIsoChannel	Isochronous channel number of the outgoing data stream. The channel number is set up by the camera controller and is stored in the *DCAMOUTs_VIDEO_DATA* structure (that was passed to *dcamoutInitializeVideoStatCtrlRegs*) by the DCAM camera API.
usPacketSize	Isochronous transmit packet size in bytes. The packet size is determined by the current format / mode / frame rate. For format 0, 1 and 2, fixed packet sizes are being used and can be queried by function *dcaminQueryPacketSize*. For format 7, available packet sizes are set up by the camera, can be queried with function *dcaminQueryCameraFormat7* and can be modified by the camera controller.
usTransmitSpeed	Isochronous transmit speed. Valid values are Speed_100, Speed_200, and Speed_400. The actual speed must be the same as set up by the camera controller and is stored in the *DCAMOUTs_VIDEO_DATA* structure (that was passed to *dcamoutInitializeVideoStatCtrlRegs*) by the DCAM camera API.
hLck	DSP/BIOS resource lock handle for IEEE1394 interface 2. Typically, *hLck* refers to a LCK object, which is defined in the DSP/BIOS configuration script (*.tcf* file) of the video processing application. *hLck* is used internally by the API to lock against multiple accesses to the hardware.
hSem	DSP/BIOS semaphore handle for generating outgoing frame buffer events. *hSem* will be posted whenever a frame has been completely transferred to the streaming core in the FPGA, so that the corresponding buffer is now empty and can be filled again. Typically, *hSem* refers to a SEM object, which is defined in the DSP/BIOS configuration script (*.tcf* file) of the video processing application. If frame buffer events are not required, *hSem* can be set to NULL.

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_INVALID_PARAMETER	Channel number, packet size or transmit speed are out of range.
DCAMOUT_ERROR_ISO_START	See chapter 7.4.

7.3.4 dcamoutIsoGetBufferState

While isochronous transmit is active, *dcamoutGetBufferState* can be used to check the state of a specific frame buffer. Typically, this is only used for diagnostic purposes and availability of a free output buffer is checked using a semaphore.

defined in
dcamoutApi.h

synopsis

```
DCAM_STATUS dcamoutIsoGetBufferState(INT32U u32Buffer, BOOL *bEmpty);
```

parameters

u32Buffer	Number of the buffer to be queried. Valid range is 0 to *u32BufferCount* - 1, where *u32BufferCount* was used as parameter for *dcamoutIsoListen*.
bEmpty	State of the specified buffer, set by *dcamoutIsoGetBufferState* after return. *TRUE* indicates that the buffer is empty and can be filled again by image processing software. After the buffer is filled again, it must be transmitted by calling *dcamoutIsoSubmitBuffer*. *FALSE* indicates that the buffer is still filled has not yet been completely transmitted.

return value

DCAMOUT_SUCCESS	Successful completion. *bEmpty* is set to the current buffer state.
DCAMOUT_ERROR_ISO_BUFSTAT	See chapter 7.4.

7.3.5 dcamoutIsoLockBuffer

Before image processing write to a frame buffer, *dcamoutIsoLockBuffer* must be called to mark the current buffer as locked. If a buffer was transmitted while its state is set to locked, an error is generated.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoLockBuffer(INT32U u32Buffer);
```

parameters

u32Buffer	Number of the buffer to be locked. Valid range is 0 to *u32BufferCount* - 1, where *u32BufferCount* was used as parameter for *dcamoutIsoListen*.

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_ISO_BUFFER	See chapter 7.4.

7.3.6 dcamoutIsoSubmitBuffer

dcamoutIsoSubmitBuffer must be called for each frame buffer after image processing for this frame buffer has been completed. This causes the buffer to be marked as filled and to be transmitted.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoSubmitBuffer(INT32U u32Buffer);
```

parameters

u32Buffer	Number of the buffer to be submitted. Valid range is 0 to *u32BufferCount* - 1, where *u32BufferCount* was used as parameter for *dcamoutIsoListen*.

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_ISO_BUFFER	See chapter 7.4.

7.3.7 dcamoutIsoStop

dcamoutIsoStop stops outgoing streaming at the next possible frame boundary. This ensures that only complete frames are transmitted. Transfer statistics are stored for later retrieval by *dcamoutIsoGetStatistics*.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoStop (void);
```

parameters
none

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_ISO_STOP	See chapter 7.4.

7.3.8 dcamoutIsoGetStatistics

During operation, transfer statistics are collected. These can be queried during or after operation to determine the number of processed frames or to check for errors.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoGetStatistics(S_DCAMOUT_STATISTICS *psStat);
```

parameters

psStat	Pointer to statistics structure which is initialized by *dcamoutIsoGetStatistics*. For a description of this structure see chapter 6.2.

return value

DCAMOUT_SUCCESS	Successful completion.
DCAMOUT_ERROR_ISO_STATISTICS	See chapter 7.4.

7.3.9 dcamoutIsoGetLastError

Internally, the isochronous functions call functions of the IEEE1394 API of the software streaming library. Error codes returned by these functions are stored and can be retrieved by dcamoutGetLastError for more detailed information about an error. The return value is the status code of the last-called internal API function.

defined in
dcamoutApi.h

synopsis
```
DCAM_STATUS dcamoutIsoGetLastError(void);
```

parameters
none

return value

DCAMOUT_SUCCESS	Successful completion.
0000 … 0FFFh	Error code from IEEE1394 API.
F000 … FFFFh	Error code from software streaming library, see [5].

7.4 Status Codes

Below is a list of status codes used by the system-specific functions. Most of these status codes are defined in the system-specific header files *dcaminIso.h* and *dcamoutIso.h*, however, application software should not include these headers files directly, but include *dcaminApi.h* and *dcamoutApi.h* which in turn include the system-specific headers. The remaining status codes are already defined and explained in [4], but are listed here together with additional error conditions that come from system-specific functions.
Please note that some more detailed error information can be obtained by functions *dcamoutIsoGetLastError* and *dcaminIsoGetLastError*, see sections TBD and TBD.

Status code	Encoding	Also used in
DCAMIN_SUCCESS	0000h	DCAM Frame Capture API
DCAMIN_ERROR_INVALID_PARAMETER	(see [3])	
DCAMIN_ERROR_ISO_INIT	1050h	n/a
DCAMIN_ERROR_ISO_RESOURCE	1051h	
DCAMIN_ERROR_ISO_START	1052h	
DCAMIN_ERROR_ISO_STOP	1053h	
DCAMIN_ERROR_ISO_BUFSTAT	1054h	
DCAMIN_ERROR_ISO_BUFFER	1055h	
DCAMIN_ERROR_ISO_STATISTICS	1056h	
DCAMOUT_SUCCESS	0000h	DCAM Camera API
DCAMOUT_ERROR_ISO_INIT	2081h	n/a
DCAMOUT_ERROR_ISO_START	2082h	
DCAMOUT_ERROR_ISO_STOP	2083h	
DCAMOUT_ERROR_ISO_BUFSTAT	2084h	
DCAMOUT_ERROR_ISO_BUFFER	2085h	
DCAMOUT_ERROR_ISO_STATISTICS	2086h	
DCAMOUT_ERROR_UNSUPPORTED_ISO_CHANNEL	(see [4])	DCAM Camera API
DCAMOUT_ERROR_INVALID_ISO_PACKET_SIZE		
DCAMOUT_ERROR_UNSUPPORTED_ISO_SPEED		

Table 3: Status codes summary

7.4.1 DCAMIN_ERROR_INVALID_PARAMETER

Returned by *dcaminIsoListen* if

- parameter *eMode* is not *OneChannel* or *AllChannel*
- parameter *iFirstChannelNo* or *iSecondChannelNo* is out of the allowed range of 0 to 63.

7.4.2 DCAMIN_ERROR_ISO_RESOURCE

Resource allocation failed because of one of the following reasons:

- Number of resource allocations exceeds MAX_NO_OF_CAMERAS
- Total bus bandwidth has been exceeded
- No more free channels

- Requested channel is already occupied

Resource de-allocation failed because of one of the following reasons:

- The specified channel was not found in the resource list (may have been de-allocated previously)
- An error was returned by the streaming software package.

7.4.3 DCAMIN_ERROR_ISO_START

Opening or starting streaming returned an error. This should never happen.

7.4.4 DCAMIN_ERROR_ISO_STOP

Stopping streaming failed. Most probable reasons are:

- Streaming was already stopped
- Streaming didn't stop in time (e.g. when the camera aborted image data transmit)

7.4.5 DCAMIN_ERROR_ISO_BUFSTAT

An error was returned by the streaming software package while querying a frame buffer status. Possible reasons are:

- buffer index is out of range
- Streaming operation was not started/is already stopped

7.4.6 DCAMIN_ERROR_ISO_BUFFER

An error was returned by the streaming software package while releasing a frame buffer. Possible reasons are:

- Buffer index is out of range
- Streaming operation was not started/is already stopped

7.4.7 DCAMIN_ERROR_ISO_STATISTICS

An error was returned by the streaming software package. This happens only if the interface was not yet opened or *psStat* is a nullpointer.

7.4.8 DCAMOUT_ERROR_ISO_INIT

An error was returned by the streaming software package during initialization. Possible reasons is that streaming was already initialized and opened.

7.4.9 DCAMOUT_ERROR_ISO_START

An error was returned by the streaming software package when setting up isochronous transmit operation. Most probable reasons are:

- Streaming has already been set up (*dcamoutIsoTalk* called more than once)
- Some resource isn't available (due to an aborted previous operation). If this error happens during development, the C6412Compact must be initialized using the GEL file before the software is loaded and executed.

7.4.10 DCAMOUT_ERROR_ISO_STOP

Stopping streaming failed. Most probable reasons are:

- Streaming was already stopped
- Streaming didn't stop in time (e.g. due to a missing cycle master on IEEE1394 interface 2)

7.4.11 DCAMOUT_ERROR_ISO_BUFSTAT

Status query returned an error and *bEmpty* is set to *FALSE*. Possible reasons are:

- Buffer index is out of range
- Streaming operation was not started/is already stopped

7.4.12 DCAMOUT_ERROR_ISO_BUFFER

Lock:
Buffer number out of range or interface is currently closed (*dcamoutIsoTalk* wasn't called).

Submit:
An error was returned by the streaming software package. Possible reasons are an invalid buffer number or an unopened / already closed interface.

7.4.13 DCAMOUT_ERROR_ISO_STATISTICS

An error was returned by the streaming software package. This happens only if the interface was not yet opened or *psStat* is a nullpointer.

7.4.14 DCAMOUT_ERROR_UNSUPPORTED_ISO_CHANNEL

When setting up isochronous transmission, the specified channel number exceeds the allowed range of 0…63. Please note that the allowed range may further be limited by the DCAM Camera API.

7.4.15 DCAMOUT_ERROR_INVALID_ISO_PACKET_SIZE

When setting up isochronous transmission, the specified packet size exceeds the allowed range of 0 … 4096 bytes. Please note that the allowed range may further be limited by the DCAM Camera API.

7.4.16 DCAMOUT_ERROR_UNSUPPORTED_ISO_SPEED

When setting up isochronous transmission, the specified transmit speed exceeds the allowed range of 400Mbps. Please note that the allowed range may further be limited by the DCAM Camera API.

8 FPGA Development Support

Development of FPGA-based image processing is supported by an optional FPGD development package. Besides other features, it includes the micro-line® busmaster FPGA as a project, so that it can be modified and expanded by the customer. This allows adding high-speed image pre-processing in the FPGA.

9 List of abbreviations used in this document

API	application programming interface
BSP	board support package: a combination of software and FPGA design that provides further functionality to the C6412Compact
CCS	Code Composer Studio –TI's development environment
CPU	Central Processing Unit = processor
CSL	chip support library – a library with DSP-specific functions provided by TI
DCAM	digital camera
DMA	direct memory access – a fast data transfer method
e.g.	exempli gratia (Latin) = for example
EDMA	enhanced DMA – a specific kind of DMA used in C6000 DSPs
FFS	flash file system – a proprietary file system residently installed on Orsys CPU boards
FIFO	first-in first-out – a specific kind of sequential memory
FPGA	field programmable gate array
i.e.	id est (Latin) = that is
IEEE	Institute of Electrical and Electronics Engineers, Inc.
IEEE1394	specification of a high-speed serial bus
IIDC	Instrumentation and Industrial Control Working Group (II-WG) of the 1394 Trade Association, Digital Camera Sub Working Group (DC-SWG)
JTAG	Joint Test Action Group – name of an interface for testing and debugging hardware
LED	light emitting diode
Mbps	transfer speed in bits per second; 1Mbps = 1000kbps = 10^6 bps
PDT	peripheral device transfer – a special EDMA transfer method the of the TMS320C6412DSP
SDRAM	synchronous dynamic random access memory
TBC	to be changed = value not 100% tested and may change in future
TBD	to be defined = value is not yet specified
TI	Texas Instruments
UART	universal asynchronous receiver transmitter
VGA	Video graphics adapter – used in this document to refer to a specific set of image resolutions
YUV	a specific color coding scheme with separate intensity (Y) and color (U, V) information.

10 Literature references

Further information that is not covered in this user's guide can be found in the documents listed below. References to this list are given in square brackets throughout this document. The documents are listed by title, author and literature number or file name. Additionally, a reference about DSP-related documents and other components of the C6412Compact can be found in [8].

[1] *Power Supply Board*, Orsys, `power_supply`
[2] *Flash File System User's Guide*, Orsys, `FFS_ug`
[3] *DCAM Frame Capture API User's Guide*, Orsys, `DCAM_FC_API_ug`
[4] *DCAM Camera API User's Guide*, Orsys, `DCAM_Camera_API_ug`
[5] *C6412Compact Software Streaming Package User's Guide*, Orsys, `C6412CPT_sw_str_UG.pdf`
[6] *C6412Compact micro-line® Busmaster Board Support Package*, Orsys, `C6412Cpt_ml_bm_ug`
[7] *C6412Compact DM Port Streaming Cores User's Guide*, Orsys, `C6412Cpt_str_core_ug`
[8] *C6412Compact board library user's guide*, Orsys, `C6412Cpt_Boardlib_ug`
[9] *C6412Compact Hardware Reference Guide*, Orsys, `C6412Cpt_hrg`
[10] *Texas Instruments website at www.ti.com*
[11] *TMS320 DSP/BIOS User's Guide*, TI, `SPRU423`
[12] *TMS320 DSP/BIOS API Reference Guide*, TI, `SPRU403`
[13] *TMS320C6000 Chip Support Library API Reference Guide*, TI, `SPRU401`
[14] *TMS320C64x Image/Video Processing Library Programmer's Reference*, TI, `SPRU023`
[15] *IIDC 1394-based Digital Camera Specification Version 1.30*, 1394 Trade Association, `1999023`
[16] *IEEE Standard for a High Performance Serial Bus*, IEEE, `IEEE Std 1394-1995`
[17] *IEEE Standard for a High Performance Serial Bus—Amendment 1*, IEEE, `IEEE Std 1394a-2000`
[18] *IEEE Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses*, IEEE, `IEEE Std 1212-2001`